**Programación Estructurada**

**UNIDAD I: Conceptos Básicos**

El programa es la fuerza conductora de cualquier tarea que hace una computadora. Un programa es una lista de instrucciones detalladas que indican a la computadora lo que ha de hacer. La computadora no puede hacer nada sin un programa. Es el trabajo del programador de escribir programas lo que influyen en la computadora y tomar datos en brutos y transformarlos en información significativa para el usuario final.

Los programas modernos producen información en muchos formatos. La salida se envía a pantalla y a la impresora que constituyen los dispositivos más usuales para acceso a la computadora; es decir; los datos de entrada y los datos de salida pueden ser, realmente, cualquier cosa, texto, dibujos, sonidos, etc.

HARDAWARE

Cuando un usuario interactúa con una computadora, proporciona una “entrada”; en respuesta, la computadora procesa la entrada devolviendo una “salida” valiosa al usuario. Una computadora necesita disponer de un conjunto de funcionalidades y proporcionar la capacidad de:

1. Aceptar la entrada
2. Visualizar o presentar la salida
3. Almacenar la información en un formato consistente lógicamente
4. Ejecutar operaciones aritméticas o lógicas bien sobre datos de entradas (binarios)
5. Monitorizar, controlar y dirigir las operaciones globales y de secuencia del sistema

Almacenamiento secundario

Memoria

Salida

Entrada

**UC**

Unidad de control

**Procesador**

(Unidad Central de Proceso)

**UAL**

Unidad Aritmética y Lógica

* **El procesador**

El procesador es el dispositivo interior de la computadora que ejecuta las instrucciones del programa. Otros términos sinónimos utilizados para nombrar al procesador son **UCP** (Unidad Central de Procesamiento). El procesador solo puede ejecutar instrucciones simples, tales como cálculos aritméticos sencillos o desplazamientos de números por diferentes posiciones. Sin embargo, la velocidad a la cual se realizan estos cálculos deben ser muy grande y esta característica le permite ejecutar instrucciones que realizan cálculos complejos.

La **UCP** consta de dos componentes; la “unidad de control” (CU), que coordina las actividades de la computadora y determina que operaciones se deben realizar y en que orden; asimismo controla y sincroniza todo el proceso de la computadora. La “unidad aritmético-lógica” (ALU); que realiza operaciones aritméticas y lógicas, tales como suma, resta, multiplicación, división y comparaciones. Los datos en la memoria central se pueden leer (recuperar) o escribir (cambiar) por la UCP.

* **El microprocesador**

El microprocesador es un chip (circuito integrado) que controla y realiza las funciones y operaciones con los datos. En realidad, el microprocesador representa a la Unidad Central de Proceso o procesador. La velocidad de un microprocesador se mide en megahercios (MHz) y manipulan palabras de 4 a 64 bits.

* **Dispositivos de Entrada/Salida (E/S)**

Para que el usuario pueda introducir la entrada, la computadora tiene varios “dispositivos de entrada” como parte de su hardware; el teclado, y el ratón son los más populares. La salida al usuario se le proporciona mediante” dispositivos de salida”, tales como impresora o pantalla. Los datos se leen de los dispositivos de entrada y se almacenan en la memoria central o interna. Los dispositivos de entrada convierten la información de entrada en señales eléctricas que se almacenan en la memoria central.

El teclado y la pantalla constituyen, en muchas ocasiones, un único dispositivo denominado “terminal”. En ocasiones a la impresora se la conoce como “dispositivo de copia dura”, debido a que la escritura en la impresora es una copia permanente de la salida, y a la pantalla se la denomina en contraste: “dispositivo de copia blanda”, ya que se pierde la pantalla actual cuando se visualiza la siguiente.

Los dispositivos de entrada/salida y los dispositivos de almacenamiento secundario o auxiliar se conocen como “dispositivos periféricos o simplemente periféricos” ya que normalmente son externos a la computadora.

* **La memoria central (interna)**

La memoria central o simplemente memoria se utiliza para almacenar información. Se divide a su vez en memoria RAM y memoria ROM. La memoria RAM (Random, Access Memory) es normalmente volátil, lo que significa que todo cuando se almacena o guarda en ella se pierde cuando se apaga la computadora. En general, la información almacenada en memoria puede ser de dos tipos: las instrucciones de un programa y los datos con los que operan las instrucciones. La memoria RAM puede ser *estática o dinámica* según sea el proceso de fabricación. Las memorias actuales se clasifican en DRAM, SDRAM, EDRAM, etc.

La memoria ROM (Read Only Memory) memoria de solo lectura, contiene instrucciones fundamentales que no se pueden modificar o perder de modo accidental por el usuario. Estas memorias incluyen aquellas instrucciones que son necesarias para cargar inicialmente el software cuando se enciende la maquina y cualesquier otras instrucciones que el fabricante requiere para estar accesible desde un modo personal cuando el computador está funcionando. La ROM es no volátil, su contenido no se pierde cundo se apaga la energía. Al ser esta memoria solo lectura, los programas almacenados en los chips ROM no se pueden modificar y suelen utilizarse para almacenar los programas del sistema que sirven para arrancar la computadora.

SOFTWARE: LOS ALGORITMOS

Un algoritmo es un método para resolver un problema. La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto

Programa de computadora

Diseño del algoritmo

Problema

Los pasos para la resolución de un problema son:

1. Diseño del algoritmo que describe la secuencia ordenada de pasos, que conducen a la solución de un problema dado (análisis del problema y desarrollo del algoritmo).
2. Expresar el algoritmo como un programa en un lenguaje de programación adecuado (fase de codificación).
3. Ejecución y validación del programa por la computadora.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta: sin embargo, el algoritmo será siempre el mismo.

El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la programación. En esencia, la solución de un problema se puede expresar mediante un algoritmo.

**Características de los algoritmos**

Las características fundamentales son:

* Un algoritmo debe ser preciso e indicar el orden de realización de cada paso
* Un algoritmo debe estar definido. Si se sigue el algoritmo dos veces, se debe tener el mismo resultado
* Un algoritmo debe ser finito, si se sigue un algoritmo, se debe terminar en algún momento.

La definición de un algoritmo debe describir tres partes:

* Entrada
* Proceso
* Salida

SOFTWARE (LOS PROGRAMAS)

El software se divide en dos grandes grupos; el software del sistema y software de aplicación

El **software del sistema** es el conjunto de programas indispensables para que la maquina funcione; se denominan también programas del sistema. Estos programas son, básicamente, el sistema operativo, los editores de texto, los compiladores/interpretes y los programas de utilidad.

Los lenguajes de programación sirven para escribir programas que permitan la comunicación usuario/maquina. Unos programas especiales llamados “traductores” (compiladores o interpretes) convierten las instrucciones escritas en lenguajes de programación, en instrucciones escritas en lenguajes maquinas (0, 1 bits) que esta pueda entender

Los programas que realizan tareas concretas, nominas, contabilidad, análisis estadístico, etc., es decir, los programas que podrá escribir en C , se denominan “programas de aplicación”

Se debe diferenciar entre el acto de crear un programa y la acción de la computadora cuando ejecuta las instrucciones del programa. La creación de un programa se hace inicialmente en papel y a continuación se introduce en la computadora y se convierte en lenguaje entendible por la computadora

LOS LENGUJES DE PROGRAMACION

Cuando el procesador es una computadora, el algoritmo se ha de expresar en un formato que se denomina programa. Un programa se escribe en un lenguaje de programación y las operaciones que conducen a expresar un algoritmo en forma de programa se llaman programación.

Los principales tipos de lenguajes utilizados son:

* Lenguajes maquina
* Lenguajes de bajo nivel (ensamblador)
* Lenguajes de alto nivel

**Instrucciones a la computadora**

Un programa consta de una secuencia de instrucciones, cada una de las cuales especifica ciertas operaciones que debe ejecutar la computadora.

Las instrucciones básicas y comunes a casi todos los lenguajes de programación se pueden condensar en cuatro grupos

* Instrucciones de entrada/salida: instrucciones de transferencia de información y datos entre dispositivos periféricos y la memoria central
* Instrucciones aritmético-lógicas: instrucciones que ejecutan operaciones aritméticas, lógicas
* Instrucciones selectivas: instrucciones que permiten la selección de tareas alternativas en función de los resultados de diferentes expresiones condicionales
* Instrucciones repetitivas: instrucciones que permiten la repetición de secuencias de instrucciones un número determinado de veces

**Lenguaje maquina**

Los lenguajes maquina son aquellos que están escritos en lenguajes directamente inteligibles por la computadora, ya que sus instrucciones son cadenas binarias que especifican una operación y las posición de memoria implicadas en la operación se denominan instrucciones de maquina o código maquina.

Las instrucciones de lenguaje maquina dependen del hardware de la computadora y, por lo tanto, diferirán de una computadora a otra

Las ventajas de programar en lenguaje maquina son la posibilidad de cargar (transferir un programa a la memoria) sin necesidad de traducción posterior, lo que supone una velocidad de ejecución superior cualquier otro lenguje de programación

Los inconvenientes son; dificultad y lentitud en la codificación, poca fiabilidad, dificultad grande d e verificar y poner a punto los programas, los programas solo son ejecutables en el mismo procesador

**Lenguaje de bajo nivel**

El lenguaje de bajo nivel por excelencia es el “ensamblador”. Las instrucciones en lenguaje ensamblador son instrucciones conocidas como **nemotécnicas**

Un programa escrito en lenguaje ensamblador no puede ser ejecutado directamente por la computadora, sino que requiere una frase de traducción al lenguaje maquin**a**

El programa original escrito en lenguaje ensamblador se denomina “programa fuentes” y el programa traducido en lenguaje maquina se conoce como “programa objeto”

El traductor de programas fuentes a objeto es un programa llamado “ensamblador”, existente en todas las computadoras.

El programa ensamblador, encargado de efectuar la traducción del programa fuente escrito a lenguaje maquina, con el lenguaje ensamblador, lenguaje de programación con una estructura y gramática definidas.

Los lenguajes ensambladores presentan la ventaja frente a los lenguajes maquinas de su mayor facilidad de codificación y, en general, su velocidad de cálculo.

Los inconvenientes más notables de los lenguajes ensambladores son

* Dependencia total de la maquina, lo que impide la transportabilidad de los programas
* La formación de los programas es más compleja que la correspondiente a los programas de alto nivel, ya que exige no solo las técnicas de programación, sino también el conocimiento del interior de la maquina

**Lenguaje de alto nivel**

Están diseñados para que las personas escriban y entiendan los programas de un modo mucho mas fácil que los lenguajes maquina y ensambladores. Otra razón es que un programa escrito en lenguaje de alto nivel es independiente de la maquina; las instrucciones del programa de computadora no dependen del diseño del hardware o de una computadora en particular. En consecuencia, los programas escritos en lenguaje de alto nivel son portables o transportables, lo que significa la posibilidad de poder ser ejecutados con poca o ninguna modificación en diferentes tipos de computadora.

Las siguientes ventajas son:

* El tiempo de formación de los programadores es relativamente corto
* La escritura de programas se basan en reglas sintácticas similares a los lenguajes humanos
* Las modificaciones y puestas a punto de los programas son mas fáciles
* Reducción del coste de los programas
* Transportabilidad

Los inconvenientes son:

* Incremento del tiempo de puesta a punto, al necesitarse diferentes traducciones del programa fuente para conseguir el programa definitivo.
* No se aprovechan los recursos internos de la maquina
* Aumento de la ocupación de memoria
* El tiempo de ejecución de los programas en mucho mayor

**Traductores de lenguaje**

* Intérprete: es un traductor que toma un programa fuente, lo traduce y a continuación lo ejecuta
* Compilador: es un programa que traduce los programas fuentes escritos en lenguaje de alto nivel a lenguaje maquina. Los programas escritos en lenguaje de alto nivel se llaman programas fuentes y el programa traducido programa objeto o código objeto

**La compilación y sus faces**

Para conseguir el programa maquina real se debe utilizar un programa llamado montador o enlazador (link). El proceso de ejecución de un programa escrito en un lenguaje de programación y mediante un compilador suele tener los siguientes pasos

1. Escritura del programa fuente con un editor y guardarlo en un dispositivo de almacenamiento.
2. Introducir el programa fuente en memoria.
3. Compilar el programa con el compilador C.
4. Verificar y corregir errores de compilación.
5. Obtención del programa objeto.
6. El enlazador (link) obtiene el programa ejecutable.
7. Se ejecuta el programa y, si no existe errores, se tendrá la salida del programa.

FASES EN LA RESOLUCION DE PROBLEMAS

Las fases de resolución de un problema con computadora son:

* **Análisis del problema**

Esta fase requiere una clara definición, donde se contemple exactamente lo que debe hacer el programa y el resultado o solución dada.

Para poder definir bien un problema es conveniente responder a las siguientes preguntas:

¿Qué entradas se requieren? (tipo y cantidad)

¿Cuál es la salida deseada? (tipo y cantidad)

¿Qué métodos produce la salida deseada?

* **Diseño del algoritmo**

En la etapa de diseño se determina “como” hace el programa la tarea solicitada, es decir, la resolución de un problema complejo se realiza dividiendo el problema en sub-problemas y a continuación dividir estos sub-problemas en otros de nivel más bajos, hasta que pueda ser implementada una solución en la computadora. Este método se conoce técnicamente como diseño descendente (top down) o modular. El proceso de romper el problema en cada etapa y expresar cada paso en forma más detallada se denomina “refinamiento sucesivo”.

Cada sub-programa es resuelto mediante un modulo (sub-programa) que tiene un solo punto de entrada y un solo punto de salida.

Los programas estructurados de esta forma se dice que tienen un diseño modular y el método de romper el programa en módulos mas pequeños se llama “programación modular”. Los módulos pueden ser planeados, codificados, comprobados y depurados independientemente y a continuación combinarlos entre si. El proceso implica la ejecución de los siguientes pasos hasta que el programa se termina

1. Programar un modulo
2. Comprobar el modulo
3. Si es necesario, depurar el modulo
4. Combinar el modulo con los módulos anteriores

El proceso que convierte los resultados del análisis del problema en un diseño modular con refinamiento sucesivos que permiten una posterior traducción a un lenguaje se denomina **diseño del algoritmo**

* **Herramientas de programación**

1. **Diagramas de flujo:** es una representación grafica de un algoritmo

**Símbolos de diagrama de flujo**

*Línea de flujo ( ):* representa el flujo secuencial de la lógica del programa

*El rectángulo ( ):* significa algún tipo de proceso, es decir, cualquier tipo de operación que pueda originar cambio de valor, formato o posición de la información almacenada en memoria, operaciones aritméticas, de transparencia, etc.

*El paralelogramo ( ):* símbolo de entrada y salida que representa cualquier tipo de introducción de datos en la memoria desde los periféricos, “entradas”, o registros de la información procesada en un periférico “salida”

*El rombo ( ):* representa decisión, indica operaciones lógicas o de comparación entre datos y en función del resultado de la misma cual de los distintos caminos alternativos del programa se debe seguir.

*La terminal ( ):* representa el comienzo y final de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en un programa

1. **Pseudocódigo:** es una herramienta de programación en la que las instrucciones se escriben en palabras similares al ingles o español que facilitan tanto la escritura como la lectura de programas.

* **Codificación de un programa**

La codificación es la escritura en un lenguaje de programación de la representación del algoritmo desarrollado en las etapas precedentes. Dado que el diseño de un algoritmo es independiente del lenguaje de programación utilizado para su implementación, el código puede ser escrito con igual facilidad en un lenguaje a otro

**Documentación interna:** la documentación interna es la que se incluye dentro del código del programa fuente mediante comentarios que ayudan a la compresión del código. Todas las líneas de programas que comiencen con un símbolo “/\*” son comentarios. Estas líneas de comentarios solo sirven para hacer los programas más fáciles de comprender

* **Compilación y ejecución de un programa**

Una vez que el algoritmo se ha convertido en un programa fuente, debe ser traducido a lenguaje maquina, este proceso se realiza con el compilador y el sistema operativo que se encarga prácticamente de la compilación

Si tras la compilación se presenta n errores (errores de compilación) en el programa fuente, es preciso volver a editar el programa, corregir los errores y compilar de nuevo. Este proceso se repite hasta que no se producen errores, obteniendo el programa objeto que todavía no es ejecutable directamente. Suponiendo que no existen errores en el programa fuente, se debe instruir al sistema operativo para que realice la fase de montaje o enlace (link), carga, el programa objeto con las librerías del programa del compilador

* **Verificación y depuración de un programa**

La verificación o compilación de un programa es el proceso de ejecución del programa con una amplia variedad de datos de entrada, llamados “datos de test o prueba”, que determinaran si el programa tiene errores (bugs)

La depuración es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores.

Cuando s ejecuta un programa, se pueden producir tres tipos de errores:

1. Errores de compilación: se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación y suelen ser “errores de sintaxis”
2. Errores de ejecución: estos errores se producen por instrucciones que la computadora puede comprender pero no ejecutar
3. Errores lógicos: se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo. Estos errores son los más difíciles de detectar, ya que el programa puede funcionar y no producir errores de compilación ni de ejecución, y solo puede advertir el error por la obtención de resultados incorrectos

* **Documentación y mantenimiento**

La documentación de un problema consta de la descripciones de los pasos a dar en el proceso de resolución de un problema.

La documentación de un programa puede ser interna y externa. La documentación interna es la contenida en líneas de comentarios. La documentación externa incluye análisis, diagrama de flujo y/o pseudocódigos, manuales de usuario con instrucciones para ejecutar el programa y para interpretar los resultados.

La documentación es vital cuando se desea corregir posibles errores futuros o bien cambiar el programa. Tales cambios se denominan “mantenimiento de programa”. Después de cada cambio la documentación debe ser actualizada para facilitar cambios posteriores

DISEÑO DEL ALGORITMO

La descomposición del problema en sub-problemas más simples y a continuación dividir estos sub-problemas en otros más simples que pueden ser implementados para su solución en la computadora se denomina diseño descendente.

Las ventajas más importantes del diseño descendente son:

* El problema se comprende más fácilmente al dividirse en partes más simples denominados módulos
* Las modificaciones en los módulos son más fáciles
* La comprobación del problema se puede verificar fácilmente

ESCRITURA DEL ALGORITMO

El sistema para describir un algoritmo consiste en realizar una descripción paso a paso con un lenguaje natural del citado algoritmo. Un algoritmo es un método o conjunto de reglas para solucionar un problema. En cálculos elementales estas reglas tienen las siguientes propiedades:

* Debe estar seguidas de alguna secuencia definida de pasos hasta que se obtenga un resultado coherente
* Solo puede ejecutarse una operación a la vez

**Construcción de un algoritmo:** se considera un primer borrador, dado que el pseudocódigo tiene que traducirse posteriormente a un lenguaje de programación. El pseudocódigo no puede ser ejecutado por una computadora. La ventaja del pseudocódigo es que en su uso, en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico. Es también fácil de modificar el pseudocódigo si se descubren errores o anomalías en la lógica del programa, mientras que en muchas ocasiones suele ser difícil el cambio en la lógica, una vez que esta codificado en un lenguaje de programación

**Diagrama de Nassi Schneiderman (N-S):** El diagrama de N-S también conocido como diagrama de Chapin es como un diagrama de flujo en el que se omiten las flechas de unión y las cajas son contiguas. Las acciones sucesivas se escriben en cajas sucesivas y, como en los diagramas de flujo, se puede escribir diferentes acciones en una caja

|  |
| --- |
| Nombre del algoritmo |
| Acción 1  Acción 2  Acción 3 |
| … |
| Fin |

**UNIDAD II: tipos de datos y operaciones básicas**

ELEMENTOS BASICOS DE UN ALGORITMO

Los elementos básicos constitutivos de un algoritmo o programa son:

* Palabras reservadas
* Identificadores
* Caracteres especiales
* Constantes
* Variables
* Expresiones
* Instrucciones

Existen otros elementos que forman parte de los programas:

* **Bucles:** un bucle o lazo (loop) es un segmento de un algoritmo o programa, cuyas instrucciones se repiten un número determinado de veces mientras se cumpla una determinada condición. Este mecanismo es una condición que puede ser verdadera o falsa y que se comprueba una vez a cada paso o iteración del bucle (total de instrucciones que se repiten en el bucle)

Un bucle consta de tres partes: decisión, cuerpo del bucle, salida del bucle

BUCLE

ESCRIBIR

SUMA

LERR N

SUMA

SUMA + N

SUMA O

SUMA 0

LEER N

**SI**

N = 0

**NO**

ESCRIBIR SUMA

SUMA

SUMA + N

BUCLE INFINITO

BUCLE FINITO

* **Bucle s anidados:** los bucles son anidados cuando están dispuestos de tal modo que unos son interiores a otros; los bucles son independientes cuando son externos unos a otros
* **Contadores:** los procesos repetitivos son la base del uso de las computadoras. En estos procesos se necesitan normalmente contar los sucesos o acciones internas del bucle, como pueden ser los elementos de un fichero, el número de iteraciones a realizar por el bucle, etc. Una forma de controlar un bucle es mediante un contador. Un contador es una variable cuyo valor se incremente o decrementa en una cantidad constante en cada iteración

CONT 1

**SI**

CONT >50?

**NO**

REALIZAR COPIA

CONT CONT+1

* **Acumulador:** un acumulador o totalizador es una variable cuya misión es almacenar cantidades variables resultantes de sumas sucesivas. Realiza la misma función que un contador, con la diferencia de que el incremento o decremento de cada suma es variable en lugar de constante, como en el caso del contador.

LEER K

N 0

SUMA 0

N N+1

SUMA SUMA+1

N =K?

ESCRIBIR K, SUMA

* **Decisión o selección:** cuando el programador desea especificar dos o más caminos alternativos en un algoritmo, se debe utilizar estructuras de decisión o selección. Una instrucción de decisión o selección evalúa una condición y, en función del resultado esa condición, se bifurcara a un determinado punto
* **Interruptores:** un interruptor o conmutador (switch) -a veces se les denomina indicador, bandera – es una variable que puede tomar diverso valores a lo largo de la ejecución del programa y que permite comunicar información de una parte a otra del mismo.

ESCRITURA DE ALGORITMOS/PROGRAMAS

Un algoritmo constara de dos componentes: una cabecera de programa y un bloque algoritmo.

La cabecera de programa es una acción simple que comienza con la palabra “algoritmo”. Esta palabra estará seguida por el nombre asignado al programa completo.

El bloque algoritmo es el resto del programa y consta de dos componentes o secciones: las acciones de declaración y las acciones ejecutables.

Las declaraciones definen o declaran las variables y constantes que tengan nombres. Las acciones ejecutables son las acciones que posteriormente deberá realizar la computación cuando el algoritmo convertido en programa se ejecute.

|  |
| --- |
| **Algoritmo** |
| Cabecera del programa |
| Sección de declaración |
| Sección de acciones |

* **Cabecera del programa o algoritmo:** todos los algoritmo y programas deben comenzar con una cabecera en la que se exprese el identificador o nombre correspondiente con la palabra reservada que señale el lenguaje
* **Declaración de variables:** en esta sección se declaran o describen todas las variables utilizadas en el algoritmo, listándose sus nombres y especificando sus tipos
* **Declaración de constantes numéricas:** en esta sección se declaran todas las constantes que tengan nombres. Los valores de estas constantes ya no pueden variar en el transcurso del algoritmo
* **Declaración de constante y variables carácter:** las constantes de carácter simple y cadenas de caracteres pueden ser declaradas en la sección del programa **const**, al igual que las constantes numéricas
* **Comentarios:** la documentación de un programa es el conjunto de información interna y externa al programa, que facilitara su posterior mantenimiento y puesta a punto. La documentación interna, es la que se acompaña en el código o programa fuente y se realizara base de comentarios significativos. Estos comentarios se representan con diferentes notaciones, según el tipo de lenguaje de programación. La documentación externa se acompañara en información ajena al programa y proporcionada por el programador

CONCEPTO DE PROGRAMA

Un programa de computadora es un conjunto de instrucciones que producirán la ejecución de una determinada tarea. En esencia un programa es un medio para conseguir un fin

* **Partes constitutivas de un programa:** conceptualmente un programa puede ser considerado como una caja negra. La caga negra o el algoritmo de resolución, en realidad, es el conjunto de códigos que transforman las entradas del programa (datos) en salidas (resultados)

El programador debe establecer de donde provienen las entradas al programa. Las entradas, en cualquier caso, procederán de un dispositivo de entrada. El proceso de introducir la información de entrada en la memoria de la computadora se denomina “entrada de datos, operación de lectura o acción de *leer*”

Las salidas de datos se deben presentar en dispositivos periféricos de salida: pantalla, impresoras, discos, etc. La operación de salida de datos se conoce también como escritura o acción de *escribir.*

Programa (algoritmo de resolución)

Entrada

Salida

* **Instrucciones y tipos de instrucciones:** el proceso de diseño del algoritmo o posteriormente de codificación del programa consiste en definir las acciones o instrucciones que resolverán el problema.

Las acciones o instrucciones se deben escribir y posteriormente almacenar en memoria en el mismo orden en que han de ejecutarse, es decir, en “secuencia”

Un programa puede ser lineal o no lineal, un programa es lineal si las instrucciones se ejecutan secuencialmente, sin bifurcaciones, decisión ni comparaciones. Un programas no lineal cuando se interrumpe la secuencia mediante instrucciones de bifurcación

1. Instrucciones de asignación: asigna un valor a una variable
2. Instrucciones de lectura de datos (entrada): lee datos de un dispositivo de entrada
3. Instrucciones de escritura de resultados (salida): muestra el resultada de una instrucción de entrada
4. Instrucciones de bifurcación: interrupción de la ejecución de un programa.
   * + - 1. Bifurcación incondicional: se realiza siempre que el flujo del programa pase por la instrucción sin necesidad del cumplimiento de ninguna condición
         2. Bifurcación condicional: depende del cumplimiento de una determinada condición. Si se cumple la condición

DATOS, TIPOS DE DATOS Y OPERACIONES PRIMITIVAS

Un dato es la expresión general que describe los objetos con los cuales opera una computadora. La acción de las instrucciones ejecutables de las computadoras se refleja en cambios en los valores de las partidas de datos. Los datos de entrada se transforman por el programa, después de las etapas intermedias, en datos de salida. Los tipos de datos simples son los siguientes:

* **Datos numéricos:** es el conjunto de los valores numéricos. Estos pueden representarse en dos formas distintas:
  + 1. Enteros: el tipo entero es un subconjunto finito de los números enteros. Los enteros se denominan en ocasiones números de punto o coma fija. Los números enteros máximos y mínimos de una computadora suelen ser “-32768” , “+32767”
    2. Reales: el tipo real consiste en un subconjunto de los números reales. Los números reales siempre tienen un punto decimal y pueden ser positivos o negativos
* **Datos lógicos (booleanos):** el tipo lógico es aquel dato que solo puede tomar uno de dos valores; “*verdadero” o “falso”.* Este tipo de datos se utiliza para presentar las alternativas (si/no) a determinadas condiciones
* **Datos tipos carácter y tipo cadena:** el dato tipo carácter es el conjunto finito de caracteres que la computadora reconoce un dato tipo carácter contiene un solo carácter; los caracteres pueden ser alfabéticos (A, a, B, b, C, c,….Z, Z), numéricos (1, 2,4,5,6), especiales (+, -, \*,^,…).

Una cadena (string) de caracteres es una sucesión de caracteres que se encuentran delimitado por una comilla (apostrofo) o doble comillas, según el tipo de lenguaje de programación. La longitud de una cadena de caracteres es el número de ellos comprendidos entre los separadores o limitadores. Algunos lenguajes tienen datos tipos cadenas

* **Constantes y variables:** los programas de computadora contienen ciertos valores que no deben cambiar durante la ejecución del programa. Tales valores se llaman “constantes”. De igual forma, existen otros valores que cambiaran durante la ejecución del programa; a estos valores se les llama “variables”

Una variable es un objeto o partida de dato cuyo valor puede cambiar durante el desarrollo del algoritmo o ejecución del programa. Dependiendo del lenguaje, hay diferentes tipos de variables, tales como enteras, reales, carácter, lógicas y de cadena. Una variable que es de un cierto tipo puede tomar únicamente valores de ese tipo

OPERACIONES CON CADENAS

* **Calculo de la longitud de una cadena:** las operaciones de determinación de la longitud de una cadena se representan por la función “longitud”:

Longitud (cadena)

La función longitud tiene como argumento una cadena, pero su resultado es un valor numérico

* **Comparación:** la comparación de cadena (igualdad y desigualdad) es una operación muy importante, sobre todo en las clasificaciones de datos tipo carácter, que se utiliza con mucha frecuencia en aplicaciones de proceso de datos (clasificaciones de listas, tratamiento de textos, etc.)
* **Concatenación:** la concatenación es la operación de reunir varias cadenas de caracteres en unas sola pero conservando el orden de los caracteres en cada una de ellas
* **Sub-cadenas:** permite la extracción de una parte especifica de una cadena

Sub-cadena: (cadena fuente, p, longitud)

Sub-cadena: es la cadena de la que debe extraerse una sub-cadena

P: es un número entero que corresponde a la posición inicial de la sub-cadena

Longitud: es la longitud de la sub-cadena

* **Búsqueda:** una operación frecuente para realizar con cadenas es localizar si una determinada cadena forma parte de otra cadena más grande o buscar la posición en que aparece un determinado carácter o secuencia de caracteres de un texto

OTRAS FUNCIONES DE CADENAS

* **Insertar:** si se desea insertar una cadena “c” dentro de un texto o cadena mas grane, se debe indicar la posición

Insertar (t, p, s)

**T:** texto o cadena donde se va a insertar

**P:** posición a partir de la cual se va a insertar

**S:** sub-cadena que se va a insertar

* **Borrar:** si se desea eliminar una sub-cadena que comienza en la posición “p” y tiene una longitud “n” se tiene la función borrar

Borrar (t, p, n)

**T:** texto o cadena de donde se va a eliminar una sub-cadena

**P:** posición a partir de la cual se va a borrar

**N:** longitud de la sub-cadena a eliminar

* **Cambiar:** la operación insertar trata de sustituir en un texto “t” la primera ocurrencia de una sub-cadena “S1” por otra “S2”

Cambiar (t, S1, S2)

**T:** texto donde se realizaran los cambios

**S1:** sub-cadena a sustituir

**S2:** sub-cadena

EXPRESIONES

Las expresiones son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones especiales

* **Expresiones aritméticas:** las expresiones aritméticas son análogas a las formulas matemáticas. Las variables y constantes son numéricas (real o entero) y las operaciones son las aritméticas. Los símbolos “ \*, -, \*, ^” y las palabras clave **div** y  **mod** se conocen como operadores aritméticos

**Operadores DIV y MOD:** el símbolo “/” se utiliza para la división real y el operador “div” representa la división entera. El operador “mod” representa el resto de la división entera, aunque hay lenguajes que utilizan otros símbolos como “%”.

* **Expresiones lógicas (booleanas):** las expresiones lógicas o booleana, cuyo valor es siempre verdadero o falso. Las expresiones lógicas se forma combinando constantes lógicas, variables lógicas y otras expresiones lógicas, utilizando los operadores lógicos “not, and y or” y los operadores relacionales “=, <, >, <=, >=, <>”

OPERACIÓN DE ASIGNACION

La operación de asignación es el modo de almacenar valores a una variable. La operación de asignación se representa con el símbolo u operador “🡨”. La operación de asignación se conoce como instrucción o sentencia de asignación cuando se refiere a un lenguaje de programación. El formato general de una operación de asignación es

Nombre de variable 🡨 expresión

**UNIDAD III: programación estructurada**

TENICAS DE PROGRAMACION

El diseño de un programa entraña la descomposición del problema en módulos (partes independientes) –programación modular-, la programación de cada modulo mediante métodos estructurados –programación estructuradas- y su unión posterior mediante procedimientos ascendentes o descendentes

PROGRAMACION MODULAR

cada programa contiene un modulo denominado “programa principal” que controla todo lo que sucede; se transfiere el control a sub-módulos (posteriormente se denominara subprogramas), de modo que ellos puedan ejecutar sus funciones; sin embargo, cada sub-modulo devuelve el control al modulo principal cuando se haya completado su tarea. Si la tarea asignada a cada sub-modulo es demasiado compleja, este deberá romperse en otros módulos más pequeños. Esta tarea puede ser *entrada, salida, manipulación de datos, control de otros módulos o alguna combinación de estos.*

Los módulos son independientes en el sentido en el que ningún modulo puede tener acceso directo a cualquier otro modulo, con excepción del modulo al que llama y sus propios sub-módulos. Sin embargo, los resultados producidos por un modulo pueden ser utilizados por cualquier otro modulo cuando se transfiera a ellos el control.

RAIZ

MODULO 1

MODULO 4

MODULO 3

MODULO 2

MODULO 4.1

MODULO 3.2

MODULO 3.1

MODULO 2.1

MODULO 1.2

MODULO 1.1

**Tamaño de los módulos:** aunque no exista una regla fija que permita definir el tamaño estándar del modulo, una buena aproximación es considerar módulos del tamaño físico de una página (30 o 40 líneas de instrucciones)

PROGRAMACION ESTRUCTURADA

La programación estructurada utiliza un número limitado de estructuras de control que minimizan la complejidad de los problemas y, por consiguiente, reducen los errores. La programación estructurada hace los programas mas fáciles de escribir, verificar, leer y mantener. Los programas deben estar dotados de una estructura

La programación estructurada es el conjunto de técnicas que incorporan:

* **Recursos abstractos:** descomponer un programa en términos de recursos abstractos consiste en descomponer una determinada acción compleja en función de un numero de acciones más simples, capaces de ser ejecutadas por una computadora y que constituirán sus instrucciones
* **Diseño descendente (top Down):** la metodología descendente consiste en efectuar una relación entre las sucesivas etapas de estructuración, de modo que se relacionen unas con otras mediante entradas y salidas de información. Es decir, se descompone el problema en etapas o estructuras jerárquicas, de modo que se puede considerar cada estructura desde dos puntos de vista: “*que hace y como lo hace*”
* **Estructuras básicas (teorema de la programación):** las estructuras básicas son; secuenciales, selectivas y repetitivas. Un programa se define como propio si cumple las siguientes características:

1. Posee un solo punto de entrada y uno solo de salida o fin para control del programa
2. Existen caminos desde la entrada hasta la salida que se pueden seguir y que pasan por todas partes del programa
3. Todas las instrucciones son ejecutables y no existen lazos o bucles infinitos

ESTRUCTURA SECUENCIAL

La estructura secuencial es aquella en la que una acción (instrucción) sigue a otra en secuencia. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente y así sucesivamente hasta el final del proceso

ESTRUCTURA SELECTIVA

Las estructuras selectivas se utilizan para tomar decisiones lógicas: de ahí que se suelan denominar también *“estructuras de decisión o alternativas”*

En las estructuras selectivas se evalúa una condición y en función del resultad de la misma se realiza una opción u otra. Las condiciones se especifican usando expresiones lógicas.

* **Alternativa simple (si – entonces / if – then)**

La estructura alternativa simple “si-entonces” ejecuta una determinada acción cuando se cumple una determinada condición

* Si la condición es verdadera, entonces ejecuta la acción “SI”
* Si la condición es falsa, entonces no hacer nada

Condición

no

si

Acciones

* **Alternativa doble (si-entonces-sino / if-then-else)**

La estructura anterior es muy limitada y normalmente se necesitara una estructura que permita elegir entre dos opciones o alternativas posibles, en función del cumplimiento o no de una determinada condición

Condición

si no

Acción 1

Acción 2

* **Alternativa múltiple (según-sea, caso de / case)**

La estructura de decisión múltiple evaluara una expresión que podrá tomar “n” valores distintos 1, 2, 3,4,…, n. según que elija uno de estos valores en la condición, se realizara una de las “n” acciones o lo que es igual, el flujo del algoritmo seguirá un determinado camino entre los “n” posibles

Condición

Acción n

Acción 4

Acción 3

Acción 2

Acción 1

……….

ESTRUCTURAS REPETITIVAS

La estructura que repite una secuencia de instrucciones un número determinado de veces se denomina “bucle”, y se llama “iteración” al hecho de repetir la ejecución de una secuencia de acciones.

Tales opciones repetidas se denominan “bucles o lazos”. Las acciones que se repiten en un bucle se denominan “iteración”. Las dos principales preguntas a realizarse en el diseño de un bucle son: “¿Qué contiene el bucle? Y ¿Cuántas veces se debe repetir?”

Cuando se utiliza un bucle para sumar una lista de números, se necesita saber cuántos números se han de sumar. Para ello se necesita conocer algún medio para detener el bucle

* **Estructura mientras (while)**

La estructura repetitiva mientras es aquella en que el cuerpo del bucle se repita mientras se cumpla una determinada condición

Cuando se ejecuta la instrucción **mientras,** la primer cosa que sucede es que se evalúa la condición (una expresión booleana). Si se evalúa como falsa, ninguna acción se toma y el programa prosigue en la siguiente instrucción del bucle. Si la expresión booleana es verdadera, entonces se ejecuta el cuerpo del bucle, después de lo cual se evalúa de nuevo la expresión booleana. Este proceso se repite una y otra vez mientras la expresión booleana sea verdadera

1. **Ejecución de un bucle cero veces:** en una estructura mientras la primera cosa que sucede es la evaluación de la expresión booleana; si se evalúa “falsa” en ese punto, entonces el cuerpo del bucle nunca se ejecutara
2. **Bucle infinito:** algunos bucles no exigen fin y otros no encuentran el fin por error en su diseño. Un bucle que nunca se termina se denomina bucle infinito o sin fin, los bucles sin fin no intencionados son perjudiciales para la programación
3. **Terminación de bucles con datos de entrada:** si su algoritmo o programa esta leyendo una lista de valores con un bucle “mientras”, se debe incluir algún tipo de mecanismo para terminar el bucle; *“preguntar antes de la iteración”, “encabezar la lista de datos con su tamaño”, “finalizar la lista con su valor de entrada”, “agotar los datos de entradas”.* Tal vez el método mas correcto para terminar un bucle que lee una lista de valores es con un centinela. Un valor centinela es una valor especial usado para indicar el final de una lista de datos

* **Estructura repetir (do/while)**

La estructura repetir se ejecuta hasta que se cumpla una condición determinada que se comprueba al final del bucle.

El bucle “repetir-hasta que” se repite mientras el valor de la expresión booleana de la condición sea falsa (justo la opuesta de la sentencia mientras)

Cuando una instrucción “repetir” se ejecuta, lo primero que sucede es la ejecución del bucle y, a continuación, se evalúa la expresión booleana resultante de la condición. Si se evalúa como falsa, el cuerpo del bucle se repite y la expresión booleana se evalúa una vez. Después de cada iteración del cuerpo del bucle, la expresión booleana se evalúa; si es verdadera, el bucle termina y el programa sigue en la siguiente instrucción “hasta que”

* **Diferencias de la estructura mientras y repetir**

1. La estructura “mientras” termina cuando la condición es falsa, mientras que “repetir” termina cuando la condición es verdadera
2. En la estructura “repetir” el cuerpo del bucle se ejecuta siempre al menos una vez; por el contrario, mientras es mas general y permite la posibilidad de que el bucle pueda no ser ejecutado

* **Estructura desde/para (for)**

La estructura “desde” ejecuta las acciones del cuerpo del bucle un numero especificado de veces y de modo automático controla el numero de iteraciones o pasos a través del cuerpo del bucle

* **Otras representaciones de estructuras repetitivas desde/para**

Un bucle desde se representa con los símbolos de proceso y decisión mediante un contador

La estructura “desde” comienza con un valor inicial de la variable índice y las acciones especificadas se ejecutan a menos que el valor inicial sea mayor que el valor final. La variable índice se incrementa en uno y si este nuevo valor no excede al final, se ejecutan de nuevo las acciones. Por consiguiente, las acciones específicas en el bucle se ejecutan para cada valor de la variable índice desde el valor inicial hasta el valor final con el incremento de uno en uno

La variable índice o e control normalmente será de tipo entero y es normal emplear como nombres las letras “I, J, K”

Si el valor inicial de la variable índice es menor que el valor final, los incrementos deben ser positivos, ya que en caso contrario la secuencia de acciones no se ejecutaría. De igual modo, si el valor inicial es mayor que el valor final, el incremento debe ser en este caso negativo, es decir, decremento

SALIDAS INTERNAS DE LOS BUCLES

En ocasiones es necesario disponer de una estructura repetitiva que permita la salida en un punto intermedio del bucle cuando se cumple una condición. Las salidas de bucles suelen ser validas en estructuras , mientras, repetir y desde

* **Bucle controlado por centinela**

Crear un algoritmo para calcular el salario de una serie de trabajadores, cuyos datos serán introducidos por teclado. Sabiendo que las horas normales se pagan a 2000 y las extraordinarias a 2500 mientras no excedan de 10. Si las horas extraordinarias sobrepasan las 10, el exceso se pagaría a 2350. La introducción de datos terminara cuando se escriba “\*” como nombre para un trabajador

**Algoritmo: calcular\_salario**

**Variable:**

**Nombre: cadena Hextraordinarias: real**

**Hnormales: real Salario: real**

**Inicio**

**Escribir: “deme nombre (\*para fin)”**

**Leer: nombre**

**Mientras (nombre <> \*) hacer**

**Escribir:”ingresar cantidad de horas normales”**

**Leer: Hnormales**

**Escribir:”ingresar cantidad de horas extraordinarias”**

**Leer: Hextraordianrias**

**Salario 🡨 Hnormales\*2000**

**Si (Hextraordianrias <= 10) entonces**

**Salario 🡨 salario + Hextraordianrias\*2500**

**Sino**

**Salario 🡨 salario+10\*Hextraordinarias+(Hextraordinarias -10)\*2350**

**Fin si**

**Escribir:”el nombre y el salario son: “nombre, salario**

**Escribir:”deme nombre (\* para fin)”**

**Leer: nombre**

**Fin mientras**

**Fin**

* **Bucle controlado por bandera**

Crear un algoritmo que nos determine si n número entero leído de teclado es o no primo. Un número es compuesto en cuanto que tiene algún divisor distinto de sí mismo o la unidad

**Programa: es\_o\_no\_primo**

**Variable:**

**N: entero primo: logico**

**Divisor: entero**

**Inicio**

**Escribir:”ingresar numero”**

**Leer: n**

**Primo 🡨verdadero**

**Divisor 🡨2**

**//la variable primo controla el bucle**

**Mientras (divisor <= n div 2) y (primo=verdadero) hacer**

**Si (n mod divisor = 0) entonces**

**Primo 🡨falso**

**Si no**

**Divisor 🡨divisor+1**

**Fin si**

**Fin mientras**

**Si (primo=falso) entonces //la variable primo pasa información a otra parte del programa**

**Escribir:”el numero es primo: “n**

**Si no**

**Escribir:”el numero: “n**

**Escribir:”es divisible por: “Divisor**

**Fin si**

**Fin**

* **Bucle controlado por contador**

Crear un algoritmo que nos pide la introducción por teclado de las notas de una clase de 40 alumnos y nos calcula el tanto por ciento de suspensión

**Programa: tanto\_por\_ciento**

**Variable:**

**Contalumno: entero tantoporciento: real**

**Contsuspensos: entero n: real**

**Inicio**

**Contalumno🡨 1**

**Contsuspensos🡨 0**

**Mientras (contalumno <= 40) hacer**

**Escribir:”nombre del alumno es: “alumno, contalumno, -deme nota”**

**Leer: n**

**Si (n<5) entonces**

**Contsuspensos 🡨 contsuspensos +1**

**Fin si**

**Contalumno 🡨 contalumno+1**

**Fin mientras**

**Tantoporciento🡨contsuspensos\*100/40**

**Escribir:”el tanto por ciento de suspenso es: “tantoporciento**

**Fin**

ESTRUCTURAS ANIDADAS

Las estructuras de selección “si-entonces” y “si-entonces-sino” implican la selección de una de dos alternativas. Es posible también utilizar la instrucción “si” para diseñar estructuras de selección que contengan más de dos alternativas

Las estructuras “si” interiores a otras estructuras “si” se denominan “anidadas o encajadas”

Una estructura selectiva múltiple constara e una serie de estructuras “si”, unas interiores a otras

**UNIDAD IV: lengua de programación en C/C++**

ESTRUCTURA GENERAL DE UN PROGRAMA EN C++

Una función en C++ es un grupo de instrucciones que realizan una o más acciones. Asimismo, un programa contendrá una serie de directivas **“#include”** que permitirán incluir en el mismo archivo de cabecera que a su vez constaran de funciones y datos predefinidos en ellos

1. #include <iostream.h>
2. // el programa imprime “bienvenido a la programación C++
3. Int main ()
4. {
5. Cout<<”bienvenido a la programación en C++”;
6. Return 0;
7. }

La directiva “#include” de la primera línea es necesaria para que el programa tenga salida. Se refiere a un archivo externo denominado “iostream.h” en el que se proporciona la información relativa al objeto “cout (operación de escritura)”. Obsérvese que los ángulos “< y >” no son parte del nombre del archivo; se utilizan para indicar que el archivo es un archivo de la biblioteca estándar C++

La segunda línea es un “comentario”, identificado por dobles barras inclinadas (//). Los comentarios se incluyen en programas que proporcionan explicaciones a los lectores de los mismos

La tercera línea contiene la cabecera de la función “main () (contiene declaraciones de variables e instrucciones necesarias para controlar las operaciones que ejecuta el programa)”, obligatoria en cada programa de C++. Indica el comienzo del programa y requiere solo los paréntesis “()”

La cuarta y séptima línea contiene solo las llaves “{ y }” que encierran el cuerpo de la función “main ()” y son necesaria en todos los programa de C++

La quinta línea contiene la sentencia “cout<<”bienvenido a la programación en C++”, que indica al sistem que envi el mensaje entre comillas al objeto “cout”. Este objeto es el “flujo estándar de salida” que normalmente representa la pantalla de presentación de la computadora

* **Directivas del pre-procesador**

El pre-procesador se lo puede considerar un editor de texto inteligente que consta de directivas. Las directivas más usuales son; “#include” y “#define”

Todas las directivas del pre-procesador comienzan con el signo de libro o “almohadilla (#)”, que indica al compilador que lea las directivas antes de compilar la parte principal del programa

Las directivas son generalmente instrucciones que se dan al compilador antes de que el programa se compile

La directiva “#include” indica al compilador que lea el archivo fuente que viene a continuación de ella y su contenido lo inserte en la posición donde se encuentra dicha directiva. Estos archivos se denominan archivos de cabecera o archivos de inclusión

Los archivos de cabecera (archivos con extencion “.h” o “.hpp” contienen código fuente C/C++) se situan en un programa C/C++ mediante la directiva del procesador “#include” con una instrucción que tiene el siguiente formato:

***#include<iostream.h>***

Cuando se instala el compilador, estos archivos de cabecera se almacenaran automáticamente en su disco en el directorio de inclusión (include) del sistema. Sus nombres de archivo siempre tienen la extensión “.h”

* **Declaraciones globales**

Las declaraciones globales indican al compilador que las funciones definidas por el usuario o variables asi declaradas son comunes a todas las funciones de su programa. Las declaraciones golbales se situan antes de la función “main ()”

***//declaraciones globales***

***Int calificaciones;***

***Main ()***

***{***

***//..***

***}***

* **Función main ()**

Cada programa en C++ tiene una función “main()” que es un punto inicial de entrada al programa

***Main ()***

***{***

***…***

***}***

Las sentencias incluidas entre las llaves se denominan “bloques”. Un programa puede tener solo una función “main ()”. Si se intenta hacer dos funciones o más se producirá un error

Además de la función “main ()”, un programa consta de una colección de funciones

***“una función C++ es un subprograma que devuelve un único valor, un conjunto de valores o realiza alguna tarea específica tal como E/S”***

* **Funciones definidas por el usuario**

las funciones definidas por el usuario se invocan por su nombre y los parámetros opcionales que puedan tener. Después de que la función se ejecuta, el código asociado con la función se ejecuta y, a continuación, se retorna a la función llamadora

todas la funciones tienen nombre y una lista de valores que reciben. Se puede asignar cualquier nombre a su función, pero normalmente se procura que dicho nombre describa el propósito de la función

En C++ las funciones requieren una declaración o prototipo en el programa:

***Void contararriba (int valor);***

Una declaración de función indica al compilador el nombre de la función que se está invocando en el programa. La palabra reservada “void” significa que la función o devuelve un valor

* **Comentarios**

Un comentario es cualquier información que se añade a su archivo fuente para proporcionar información de cualquier tipo. El uso de comentarios es totalmente opcional , aun que dicho uso es muy recomendable

CREACION DE UN PROGRAMA

Para crear un programa se utilizan las siguientes etapas:

1. Definir un programa
2. Definir directivas del preprocesador
3. Definición de declaraciones globales
4. Crear main()
5. Crear el cuerpo del programa
6. Crear sus propias funciones definidas por el usuario
7. Compilar, enlazar, ejecutar y comprobar su programa
8. Utilizar comentarios

PROCESO DE EJECUCION DE UN PROGRAMA EN C++

La primera operación en el proceso de ejecución de un programa es introducir la sentencia (instrucción) del programa en un editor de texto. El editor almacena el texto y debe proporcionare un nombre tal como “nombre.cpp”. A continuación se debe guardar el texto en disco para su conservación y uso posterior

Una vez editado un programa, se le proporciona un nombre. La siguiente etapa es la de “compilación”. En ella se traduce el código fuente escrito en lenguaje C++ a código maquina. El programa que realiza esta traducción se llama compilador. Cada compilador se construye para un determinado lenguaje de programación.

Cada lenguaje de programación tiene unas reglas especiales para la construcción de programación que se denomina “sintaxis”. El compilador lee el programa del archivo de texto creado anteriormente y comprueba que el programa sigue las reglas de sintaxis del lenguaje de programación. Cuando se compila su programa, el compilador traduce el código fuente C++ (las sentencias del programa) en un código maquina (código objeto). El código objeto consta de instrucciones maquina e información de cómo cargar el programa en memoria antes de su ejecución. Si el compilador encuentra errores, los presentara en la pantalla

El archivo objeto contiene solo la traducción del código fuente. Esto no es suficiente para ejecutar realmente el programa. Es necesario incluir los archivos de biblioteca. Una biblioteca es una colección de código que ha sido programada y traducida y lista para utilizar en su programa

Por consiguiente, puede haber varias archivos objetos. Un programa especial llamado “enlazador (linker)” toma el archivo objeto y las partes necesarias de la biblioteca “iostrean” y construye un archivo ejecutable. Este archivo ejecutable contiene todo el código maquina necesario para ejecutar el programa. El proceso de ejecución de un programa no suele funcional la primera vez. El proceso de detectar y corregir errores se denomina “depuración o puesta a punto” de un programa

Programa ejecutable

Enlazador

Código fuente

Compilador

Código fuente

Biblioteca

* + Etapas del proceso
    - El código fuente (archivo del programa) se crea con la ayuda del editor de texto
    - El compilador traduce el archivo texto en un archivo objeto
    - El enlazador pone juntos a diferentes archivos objetos para poner un archivo ejecutable
    - El sistema operativo pone el archivo ejecutable en la memoria central y se ejecuta el programa

DEPURACION DE UN PROGRAMA EN C++

el proceso de encontrar errores se denomina “depuración” del programa. Existen diferentes situaciones en las cuales se suelen introducir errores en un programa

1. Violación (no cumplimiento) de las reglas gramaticales del lenguaje de alto nivel en el que se escribe el programa
2. Los errores en el diseño del algoritmo en el que esta basado el programa

**Errores de sintaxis**

Los errores de sintaxis son aquellos que se producen cuando el programa viola las reglas de gramática del lenguaje. Algunos errores típicos son:

1. Puntos y como después de la cabecera main()
2. Omisión de puntos y coma al final de una sentencia
3. Olvido de la doble barra inclinada antes de un comentario
4. Olvido de las dobles comillas al cerrar una cadena
5. Etc.

**Errores lógicos**

Un segundo tipo de error importante es el error lógico, ya que tal error representa errores del programador en el diseño del algoritmo y posterior programa. Los errores lógicos son más difíciles de encontrar y aislar, ya que no suelen ser detectados por el compilador

**Errores de regresión**

Los errores de regresión son aquellos que se crean accidentalmente cuando se intenta corregir un error lógico. Los errores de regresión son comunes, pero son fáciles de leer y corregir. Una ley no escrita es que “un error se ha producido, probablemente, por el ultimo código modificado”

**Mensajes de error**

Los mensajes de error producidos durante la compilación se suelen producir, normalmente, por errores de sintaxis y sueles varias según los compiladores

* Errores fatales: algunos de ellos indican un error interno del compilador
* Errores de sintaxis: errores de línea de órdenes y errores de acceso a memoria o disco
* Advertencias: indican condiciones que son sospechosas, pero son legitimas como parte del lenguaje

**Errores en tiempo de ejecución**

Un error en tiempo de ejecución puede ocurrir como resultado del programa que obliga a la computadora a realizar una operación ilegal tal como dividir un número por cero o manipular datos no validos o no definidos

PRUEBAS

Es preciso tener presente que el compilador puede no emitir ningún mensaje de error durante la ejecución, y eso no garantiza que el programa sea correcto. Recuerde que el compilador solo indica si se escribió bien sintácticamente el programa. No indica si el programa hace lo que realmente desea que haga. Los errores lógicos pueden aparecer por un mal diseño del algoritmo y posterior programa

Para determinar si un programa contiene un error lógico, se debe ejecutar utilizando datos de muestra y comprobar la salida verificando su exactitud. Esta prueba (testing) se debe hacer varias veces utilizando diferentes entradas, preparadas por personas diferentes al programador, que puedan indicar suposiciones no evidentes en la elección de los datos de prueba. Si cualquier combinación de entradas produce salida incorrecta, entonces el programa contiene un error lógico

LOS ELEMENTOS DE UN PROGRAMA EN C++

* **Tokens (elementos léxicos de los programas)**
  + Identificadores: un identificador es una secuencia de caracteres, letras, dígitos y subrayados. En C++ el identificador puede ser de cualquier longitud; sin embargo, el compilador ignora cualquier carácter fuera de los 32 primeros. Un consejo que puede servir es:

1. Escribir identificadores de variables en letras minúsculas
2. Constantes en mayúsculas
3. Funciones con tipo de letra mixtos
   * Palabras reservadas: una palabra reservada, tal como “void”, es una característica del lenguaje C++ asociada con algún significado especial
   * Comentarios: un comentario que comienza con “//” termina al final de la línea en que se encuentre el símbolo. Los comentarios encerrados entre “/\* y \*/” pueden extenderse a lo largo de varias líneas
   * Signos de puntuación y separadores: todas las sentencias terminan con un punto y coma (;)
   * Archivos de cabecera: un archivo de cabecera es un archivo especial que contiene las declaraciones de objetos y funciones de la biblioteca. Para utilizar objetos y funciones almacenadas en una biblioteca, un programa debe utilizar la directiva “#include” para insertar el archivo de cabecera correspondiente

TIPOS DE DATOS EN C++

Los tres tipos de datos básicos son:

* Enteros
* Números de coma flotantes (reales)
* Caracteres

**Enteros (int)**: los tipos enteros se almacenan internamente en 2 bytes ( 16 bits) de memoria

* Declaración de variables: la forma mas simple de una declaración de variable en C++ es poner primero el tipo de dato y a continuación el nombre de la variable. Si se desea dar un valor inicial a la variable, este se pone a continuación

Int valor = 99

**Tipos de coma flotante (float/doublé)**: los tipos de datos de coma (punto) flotante representan números reales que contiene una coma decimal. C++ soporta tres formatos de coma flotante; el “float” requiere 4 bytes, el “doublé” requiere 8 bytes y “long doublé” requiere 10 bytes

**Caracteres (char):** un carácter es cualquier elemento e un conjunto de caracteres predefinidos o alfabeto. La mayoría de las computadoras utilizan el conjunto de caracteres ASCII.

C++ procesa datos carácter utilizando el tipo de dato “char”. Dado que el tipo “char” almacena valores en el rango de -128 a +127, C++ proporciona el tipo “unsignedchar” para representar valores de 0 a 255

EL TIPO DE DATO BOOL

El tipo “bool” proporciona la capacidad de declarar variables lógicas, que pueden almacenar los valores verdaderos y falsos

**Simulación del tipo “bool”:** si su compilador C++ no incluye el tipo “bool”, deberá utilizar el tipo de dato “int” para representar el tipo de dato “bool”. C++ utiliza el valor entero 0 para rerpesentar falso y cualquier valor entero distinto de cero (normalmente 1) para representar verdadero

**Escritura de valores “bool”:** la mayoría de las expresiones lógicas apareen en estructuras de control que sirven para determinar la secuencia en que se ejecutan las sentencias C++

CONSTANTE

1. **Constantes literales**
   1. Constantes enteras
      1. No utilizan nunca comas ni otros signos de puntuación en números enteros o completos
      2. Para forzar un valor al tipo “long”, terminar con una letra “L” mayúscula; “1024L”
      3. Para forzar un valor al tipo “usingned” terminarlo con una letra mayúscula“U”; 4352U
   2. Constantes reales

Una constante flotante representa un número real. Si el valor de coma flotante tiene una parte decimal, se puede escribir esta después del punto decimal como; “3.1234”

Para escribir números en notación exponencial, se debe seguir la parte decimal del numero con la letra E y a continuación el exponente; “4.5E+5”

* 1. Constantes carácter

Una constante carácter (char) es un carácter del código ASCII encerrado entre comillas simples; `A`

Este método se utiliza para almacenar o imprimir cualquier carácter de la tabla ASCII por su número hexadecimal. Un carácter que se lee utilizando una barra oblicua (\) se llama secuencia o código de escape

* 1. Constantes cadenas

Una constante cadena es una secuencia de caracteres encerrados entre dobles comillas; “12 de abril”

1. **Constantes definidas (simbólicas)**

Las constantes pueden recibir nombres simbólicos mediante la directiva “#define”

1. **Constantes enumeradas**

Las constantes enumeradas permiten crear listas de elementos afines. Un ejemplo típico es una constante enumerada de listas de colores, que se puede declarar como; “enum colores {rojo, verde, azul, amarillo}”

1. **Constantes declaradas “const” y “volatile”**

El cualificador “const” permite dar nombres simbólicos a constantes a modo de otros lenguajes. C++ soporta el calificador de tipo variable “const”. Especifica que el valor de una variable no se puede modificar durante el programa; “const int semana = 7; “

La palabra reservada “volatile” actúa como “const”, pero su valor puede ser modificado no solo por el propio programa, sino también por el hardware o por el software del sistema

VARIABLES

En C++ una variable es una posicion con nombre en memoria donde se almacena un valor de un cierto itpo de dato y puede ser modificado. Las variables pueden almacenar todo tipo de datos: cadenas, numeros y estructuras. Una variable tipicamente tiene un nombre (identificador) que describe su proposito. Toda variable utilizada en un programa debe ser declarada previamente

* **Delcaracion**

Una declaracion de una variable es una sentencia que proporiciona informacion de la varialbe al compilador C++

Tipo variable

Es preciso declara las variables antes de utilizarlas. Se puede ceclarar una variable en dos lugares dentro de un programa:

* 1. **al principio de un archivo o bloque de codigo**

La variable se declara al principio del archivo en codigo fuente o bien al principio de una funcion

**#include <iostream.h> // variable al principio del archivo**

**Int Numero**

**Main()**

**{**

**Cout<<”¿Cuál es el noumero favorito?”;**

**Cin>> Numero;**

**Return 0;**

**}**

**#include <iostream.h> // variable al principio de funcion**

**…**

**Main()**

**{**

**Int i;**

**Int j;**

**…**

**}**

* 1. **en el punto de utilizacion**

es posible declarar una variable en el punto donde se vaya a utilizar. Esta propiedad se utiliza mucho en el diseño de bulces

**int j;**

**for (j=0; j < 10; j++)**

**{**

**// …**

**}**

* **inicializacion de variables**

la varialbes se pueden inicializar a la vez que se declaran, o bien, inicializarse despues de la declaracion

* **declaracion o definicion**

una declaracion introduce un nombre de un objeto o de una variable y asocia un tipo con la variable/objeto. Una definicion es una declaracion que asigna simultaneamente memoria al objeto/varialbe

DURACION DE UNA VARIABLE

La zona de un programa en la que una variable esta activa se denomina “ambito o alcance”

* **varialbes locales**

las variables locales son aquellas definidas en el interior de una funcion y son visibles asolo en esa función especifica

* 1. una variable local no puede ser modificada por ninguna sentencia externa a la función
  2. los nombres de la variables locales no han de ser únicos
  3. las variables locales de la funciones no existen en memoria hasta que se ejecuta la función

**#include <iostream.h>**

**Main()**

**{**

**Int a, b, c, suma; //variables locales**

**Int numero; //variable local**

**Cout<<”cuantos números a sumar”;**

**Cinn>> numero;**

**Suma = a + b +c;**

**}**

* **variables globales**

Las variables globales son variables que se declaran fuera de la función y por defecto (omisión) son visibles a cualquier función incluyendo “main ()”

**#include <iostream.h>**

**Int a. b. c; //declaración de variables globales**

**Main()**

**{**

**Int valor; //declaración de variable local**

**//…**

**}**

ENTRADA Y SALIDAS

* **Salida (cout)**

El operador de inserción “<<” inserta datos en el flujo “cout” que los visualiza en la pantalla del equipo

* **Entrada (cin)**

El archivo de cabecera “iostream.h” de la biblioteca C++ proporciona un flujo de entrada estándar “cin” y un operador de extracción “>>”, para extraer valores del flujo y almacenarlos en variables

OPERADORES Y EXPRESIONES

Una expresión es un elemento de un programa que toma un valor. En algunos casos puede también realizar una operación.

Las expresiones pueden ser valores constantes o variables simples, tales como “25 o z”, pueden ser valores o variables combinadas con operadores (a++, m==n, etc.), o bien pueden ser valores combinados con funciones tales como “toupper (`b`)”

OPERADOR DE ASIGNACION

El operador “=” asigna el valor de la expresión derecha a la variable situada a su izquierda; “código=3425“

OPERADORES ARITMETICOS

Los operadores aritméticos sirven para realizar operaciones aritméticas básicas. Los operadores aritméticos de C++ siguen las reglas algebraicas típicas de jerarquía o prioridad

* **Asociatividad:** la Asociatividad determina el orden en que se agrupan los operadores de igual prioridad; es decir, de izquierda a derecha o de derecha a izquierda
* **Uso de paréntesis:** los paréntesis se pueden utilizar para cambiar el orden usual de evaluación de una expresión determinada por su prioridad y Asociatividad. Las sub-expresiones entre paréntesis se evalúan en primer lugar según el modo estándar y los resultados se combinan para evaluar l expresión completa. Si los paréntesis están anidados se ejecutan en primer lugar los paréntesis mas internos

OPERADORES RELACIONALES

ANSI C++ soporta el tipo “bool” que tiene dos literales “false y true”. Una expresión booleana es, por consiguiente, una secuencia de operando y operadores que se combinan para producir uno de los valores “true” y “false”

Operadores tales como “>=” y “==” que comprueban una relación entre dos operandos se llaman operadores relacionales

Los operadores relaciones se usan normalmente en sentencias de selección (if) o de iteración (while, for), que sirven para comprobar una condición. Utilizando operadores relacionales se realizan operaciones de igualdad, desigualdad y diferencias relativas

OPERADORES DE INCREMENTACION Y DECREMENTACION

De las muchas características de C++, una de las mas útiles son los operadores de incremento “++” y decremento “—“, denominados de incrementación y decrementacion, suman o restan 1 a su argumento, respectivamente, cada vez que se aplican a una variable

OPERADORES LOGICOS

Estos operadores se utilizan con expresiones para devolver un valor verdadero o un valor falso. Los operadores lógicos se denominan también operadores booleanos. Los operadores lógicos de C++ son: not (!), and (&&) y or (||)

OPERADOR COMA

El operador coma (,) permite combinar dios o mas expresiones separadas por comas en una sola línea. Se evalúa primero la expresión de la izquierda y luego las restantes expresiones de izquierda a derecha

OPERADOR CONDICIONAL

El operador condicional “?:” es un operador ternario que devuelve un resultado cuyo valor depende de la condición comprobada. Tiene Asociatividad a derechas.

Al ser un operador ternario requiere tres operandos. El operador condicional se utiliza para reemplazar a la sentencia “if-else” lógica en algunas situaciones. El formato del operador condicional es

**“expresión\_c ? expresión\_v : expresión\_f”**

Se evalúa “expresión\_c” y su valor determina cual es la expresión a ejecutar; si la condición es verdadera se ejecuta “expresión\_v” y si es falsa se ejecuta “expresión\_f”

OPERADORES ESPECIALES (), [], Y ::

* **El operador ()**

Es el operador de llamada a funciones. Sirve para encerrar los argumentos de una función, efectuar conversiones explicitas de tipo, indicar en el seno de una declaración que un identificador corresponde a una función, resolver los conflictos de prioridad entre operadores

* **El operador []**

Sirve para designar un elemento de un array. También se puede utilizar en unión on el operador “delete”; en este caso, indica el tamaño del array a destruir

* **El operador ::**

Este operador es específico de C++ y se denomina operador de ámbito de resolución y permite especificar el alcance o ámbito de un objeto

CONVERSIONES DE TIPOS

Con frecuencia, se necesita convertir un valor de un tipo u otro sin cambiar el valor que representa

* **Conversiones implícitas:** los tipos fundamentales pueden ser mezclados libremente en asignaciones y expresiones. Las conversiones se ejecutan automáticamente: los operandos de tipo mas bajo se convierten en los timo mas alto

**Int i = 12;**

**Double x = 4;**

**X = x+i //valor de i se conviete en doublé antes de suma**

* **Conversiones explicitas:** C++ fuerza la conversión explicita de tipos mediante el operador de molde (cast)

ESTRUCTURAS DE CONTROL

La estructuras de control controlan el flujo de ejecucion de un programa o funcion , permitiendo combinar instrucciones o sentencias individuales en una simple unidad logica con un punto de entra y un punto de salida

Las instrucciones o sentencias se organizan en tres tipos de estructuras de control que sirven para controlar el flujo de la ejecucion: secuencia, selección y repeticion

LA SENTENCIA “if”

En C++ la estructura de control de selección principal es una sentencia “if”. Cuando se alcanza la sntencia “if” dentro de un programa se evalua la expresion entre parentesis que viene a continuacion de “if”. Si “expresion” es verdadera, se ejecuta “accion”; en caso contrario no se ejecuta “accion”

If (expresión) acción

SENTENCIA “if” DE DOS ALTERNATIVAS: “if-else”

Cuando se ejecuta la setencia “if-else”, se evaluza expresion. Si “expresion” es verdadera, se ejecuta “accion (1)”, y en caso contrario se ejecuta “accion (2)”

If (expresión lógica)

Sentencia (1)

Else

Sentencia (2)

SENTENCIAS “if-else” ANIDADAS

Una sentencia “if” es anidada cuando la sentencia de la rama verdadera o la fama falsa es a su vez una setnecia “if”. Una sentencia “if” anidada se puede utilizar para implementar decisiones con varias alternativas o multi-alternativas

If (condicon (1))

Sentencia (1)

Else if (condición (2))

Sentencia (2)

.

.

SENTENCIA DE CONTROL “switch”

La sentencia “Switch” es una sentencia C++ que se utiliza para seleccionar una de entre múltiples alternativas. La expresión “selector” debe ser un tipo ordinal (por ejemplo, int, char, bool, pero no float o string). Cada “etiqueta” es un valor único, constante, y que debe tomar un valor diferente de los otros. Si el valor de la expresión selector es igual a una de las etiquetas “case” entonces la ejecución comenzara con la primera sentencia de la secuencia “secuencia” y continuara hasta que se encuentra una sentencia “break” (o hasta que se encuentra el final de la sentencia de control “switch”)

Switch (selector)

{

Case etiqueta (1): sentencias (1);

Break;

Case etiqueta (2): sentencias (2);

Break;

.

.

}

ESTRUCTURA DE CONTROL : BUCLES

* **La sentencia while:** un bucle es cualquier construcción de programa que repite una sentencia o secuencia de sentencias un número de veces.la sentencia (o grupo de sentencia) que se repiten en un bloque se denomina “cuerpo” del bucle y cada repetición del cuerpo del bucle se llama “iteración” del bucle. Las dos principales cuestiones de diseño en la construcción del bucle son: ¿Cuál es el cuerpo del bucle? ¿Cuántas veces se iterara el cuerpo del bucle?

Un bucle **“while”** tiene una “condición” del bucle (una expresión lógica) que controla la secuencia de repetición. La posición de esta condición del bucle es delante del cuerpo del bucle y significa que un bucle “while” es un bucle “pretest” de modo que cuando se ejecuta el mismo, se evalúa la condición antes de que se ejecute el cuerpo del bucle

While (condición del bucle)

{

Sentencia -1;

Sentencia -2;

.

.

.

Sentencia –n

}

**“NOTA: *el cuerpo del bucle se repite “mientras” que la expresión lógica (condición lógica) sea verdadera. Cuando se evalúa la expresión lógica y resulta falsa; se termina y******se ejecuta la siguiente sentencia de programa después del cuerpo de la sentencia “while””***

* + **Terminaciones anormales de un ciclo:** un error típico en el diseño de una sentencia “while” se produce cuando el bucle solo tiene una sentencia en lugar de varias sentencias, por ejemplo;

**Contador = 1;**

**While (contador < 25)**

**Cout<<Contador<<endl;**

**Contador ++;**

Se visualizara infinitas veces el valor 1. Es decir, entra en un bucle infinito del que nunca sale porque no se actualiza (modifica) la variable de control “contador”.

La razón es que el punto y coma (;) al final de la línea “cout<< contador << endl;” hace que el bucle termine en ese punto y coma (;), aunque aparentemente el sangrado pueda dar la sensación de que el cuerpo de “while” contiene dos sentencias “cout…” y “contado++;”

El error se hubiera detectado rápidamente si el bucle se hubiera escrito correspondiente con una sangría:

**Contador = 1;**

**While (contador < 25)**

**Cout<<Contador<< endl;**

**Contador ++;**

La solución es muy sencilla utilizar las llaves de la sentencia compuesta:

**Contador = 1;**

**While (contador < 25)**

**{**

**Cout<<Contador<<endl;**

**Contador++;**

**}**

* **diseño eficiente de bucles:** los principios a considerar son: primero, analizar los requisitos de un nuevo bucle con el objetivo de determinar su inicialización, prueba (condición) y actualización de la variable de control del bucle; segundo, desarrollar patrones estructurales de los bucles que se utilizan frecuentemente
* **bucles “while” con cero iteraciones:** el cuerpo de un bucle no se ejecuta nunca si la prueba o condición de repetición del bucle no se cumple (es falsa) cuando se alcanza “while” la primera vez

**Contador = 10;**

**while (Contador > 100)**

**{**

**…**

**}**

El bucle anterior nunca se ejecutara ya que la condición del bucle es falsa la primera vez que se ejecuta

* **bucles controlados por centinelas:** cuando no se conozca con exactitud cuántos elementos de datos se procesaran antes de comenzar su ejecución, se puede introducir un único dato definido y especificado denominada “valor centinela” como ultimo dato. La condición del bucle comprueba cada dato y termina cuando se lee el valor centinela. El valor centinela se debe seleccionar como mucho cuidado y debe ser un valor que no pueda producirse como dato

**// Entrada de datos numéricos**

**// Centinela -1**

**Const int centinela = -1;**

**Cout<<”introduzca primera nota”;**

**Cin>> nota;**

**While (nota != centinela)**

**{**

**Cuenta++;**

**Suma += nota;**

**Cout<<”introduzca la siguiente nota: “;**

**Cin>> nota;**

**} //fin de while**

**Cout<< “final”**

* **bucles controlados por indicadores (banderas):** las variables tipo “bool” se utilizan con frecuencia como “indicadores” o “banderas de estado” para controlar la ejecución de un bucle. El valor del indicador se inicializa (normalmente a falso) antes de la entrada al bucle y se redefine (normalmente a verdadero) cuando un suceso especifico ocurre dentro del bucle

**char car;**

**digito\_leido = false; //no se ha leído ningún dato**

**while (!digito\_leido)**

**{**

**Cout<<”introduzca un carácter: “**

**Cin>> car;**

**Digito\_leido = ((`0` <= car) && (car <= `9`));**

**…**

**} // fin de while**

El bucle funciona de la siguiente forma

* 1. entrada del bucle: la variable “digito\_leido” tiene por valor “falso”
  2. la condición del bucle “!digito\_leido” es verdadera, por consiguiente se ejecutan las sentencias del interior del bucle
  3. se introduce por teclado un dato que se almacena en la variable “car”. Si el dato leído es un cracter la variable “digito\_leido” se mantiene con valor falso, ya que ese es el resultado de la sentencia de asignación
  4. el bucle se termina cuando se lee un dto tipo (`0`a `9`) ya que la condición del bucle es falsa
* **la sentencia “break” en los bucles:**  la sentencia “break” se utiliza para realizar una terminación anormal del bucle, dicho de otro modo, la sentencia “break” se utiliza para realizar la salida de un bucle “while” o “do-while”, aunque también se puede utilizar dentro de una sentencia “switch”
* **repetición: el bucle “for”:** la sentencia “for” (bucle for) es un método para ejecutar un bloque de sentencias un número fijo de veces. El bucle “for” se diferencia del bucle “while” en que las operaciones de control del bucle se sitúan en un solo sitio: la cabecera de la sentencia

For (inicialización; condición\_iteracion; incremento)

Sentencia

1. parte de inicialización: inicializa las variables de control del bucle, se pueden utilizar variables de control del bucle simples o múltiples
2. partes de iteración: que contiene una expresión lógica que hace que el bucle realice las iteraciones de las sentencias, mientras que la expresión sea verdadera
3. parte de incremento: que incrementa o decrementa la variable o variables de control del bucle
4. sentencia: acciones o sentencias que se ejecutaran por cada iteración del bucle

* **precauciones en el uso “for”:** un bucle “for” se debe construir con gran precaución, asegurándose de que la expresión de inicialización, la condición del bucle y la expresión de incremento harán que la condición del bucle se convierta en falsa en algún momento. En particular “si el cuerpo de un bucle de conteo modifica los valores de cualquier variable implicada en la condición del bucle, entonces el numero de repeticiones se puede modificar”

**int limite = 1;**

**for (int i = 0; I <= limite; i++)**

**{**

**Cout<<i<<endl;**

**Limite++**

**}**

* **bucles infinitos:** el uso principal de un bucle “for” es implementar bucles de conteo en el que el numero de repeticiones se conoce por anticipado. Sin embargo, existen muchos problemas en los que el numero de repeticiones no se pueden determinar por anticipado.“for” permite la implementación de bucles infinitos, que son aquellos bucles que no tienen fin

For ( ; ; )

Sentencia

“la sentencia” se ejecuta indefinidamente a menos que se utilice una sentencia “return” o break”. La razón de que el bucle se ejecute indefinidamente es que se ha eliminado la expresión de inicialización, la condición del bucle y la expresión de incremento; al no existir una condición de bucle que especifique cual es la condición para terminar la repetición de sentencias, estas se ejecutaran indefinidamente

* **sentencias nulas en bucles “for”:** para ejecutar esta acción se utiliza el punto y coma (;) para marcar la sentencia vacía

**// Listado**

**// bucles “for “ con sentencias nulas**

**#include<iostream>**

**Int main()**

**{**

**Int contador = 0;**

**For ( ; contador < 5; )**

**{**

**Contador ++;**

**Cout<<”bucle!”;**

**}**

**Cout<<”\n contador: “ << contador “\n”;**

**Retunr 0;**

**}**

* **sentencias “break” y “continue”:** las sentencias “break” y “continue” terminan la ejecución de un bucle “while” de modo similar a los otros bucles

**//listado**

**//sentencias bucles “for” vacios**

**#include <iostream.h>**

**Int main ()**

**{**

**Int contador = 0;**

**Int max; //inicializacion**

**Cout<<”cuantos holas?”;**

**Cin >> max;**

**For ( ; ; ) // bucle for que no termina nunca**

**{**

**If (contador < max) // test**

**{**

**Cout << “hola!\n”;**

**Contador ++; //test**

**}**

**Else**

**Break;**

**}**

**Retunr 0;**

**}**

* **repetición: el bucle do-while:** la sentencia ç2do-while” se utiliza para especificar un bucle condicional que se ejecuta al menos una vez. Esta situación se suele dar en algunas circunstancias en las que se ha de tener las seguridad de que una determinada acción se ejecutara una o varias veces, pero al menos una vez

Do

Sentencia

While (expresión)

**#include <iostream>**

**Int main ()**

**{**

**Char opción;**

**Do**

**{**

**Cout<<”hola”<<endl;**

**Cout<<”¿desea otro tipo de saludo?\n”;**

**<<”Pulse s para si y n para no, \n”;**

**<<”y a continuación pulse intro: “;**

**Cin>> opción;**

**}while (opción == `s` || opción == `S`);**

**Cin<<”adiós;**

**Return 0;**

**}**

**UNIDAD V: programación modular**

CONCEPTO DE FUNCION

C++ se puede utilizar como lenguaje de programación estructurada, también conocida como “programación modular”, que consiste en que el programa se divida en varios módulos, en lugar de uno solo largo. Así, pues, un programa C++ se compone de varias funciones, cada una de las cuales realizan una tarea principal, por ejemplo, si está escribiendo un programa que obtenga una lista de caracteres del teclado, los ordene alfabéticamente y los visualice por pantalla, se puede escribir de la siguiente manera

**Main ()**

**{**

**Obtener\_caracteres (); //llamada a una función que obtiene los numero**

**Alfabetizar (); //llamada a la función que ordena**

**Verletras (); //llamada a la función que visualiza**

Cada función realiza una determinada tarea y cuando se ejecuta “return” se retorna al punto que fue llamado por el programa o función principal

**Return 0; //retorno a DOS**

**}**

**Void Obtener\_caracteres ()**

**{**

**//…**

**// Código de C++ para obtener una lista de caracteres**

**Return 0; // Retorno a main ()**

**}**

**Void Alfabetizar ()**

**{**

**//…**

**// Codigo de C++ para alfabetizar los caracteres**

**//…**

**Return 0; // Retorno a main ()**

**}**

**Void Verletra ()**

**{**

**//…**

**// Codigo de C++ para visualizar lista alfabetizar**

**//…**

**Return 0; // Retorno a main ()**

**}**

ESTRUCTURA E UNA FUNCION

Las funciones no se pueden anidar. Esto significa que una función no se puede declarar dentro de otra función. La razón para esto es permitir un acceso muy eficiente a los datos. En C++ todas las funciones son externos o globales, es decir, pueden ser llamadas desde cualquier punto el programa

Lista de parámetros

Tipo de resultado

Float suma (float num1, float num2)

{

Float resp;

Resp = num1+num2;

Reurn resp;

}

Cabecera de la función

Declaración de variable

Valor devuelto

Los aspectos más sobresalientes en el diseño de una función son:

* **tipo de resultado:** es el tipo de dato que devuelve y aparece antes del nombre de la función
* **lista de parámetros:** es una lista de parámetros “tipificados”
* **cuerpo de la función:** se encierra entre llaves de apertura y cierre ( {} ). No hay punto y coma (;) después de la llave de cierre
* **paso de parámetros:** se puede hacer por valor y por referencia
* no se pueden declarar funciones anidadas
* **declaración local:** las constantes, tipos de datos y variables declaradas dentro de la función son locales a la misma y no perduran fuera de ella
* **valor devuelto por la función:** mediante la palabra reservada “return” se puede devolver el valor de la función

**Nombre de una función:** un nombre de una función comienza con una letra o un subrayado y puede contener tantas letras, números o subrayados como desee

**Tipo de dato de retorno:** si la función no devuelve un valor “int”, se debe especificar el tipo de dato devuelto por la función. Si una función no devuelve un resultado, se puede utilizar el tipo “void” que se considera como un tipo de dato especial.

Muchas funciones no devuelven resultados. La razón es que se utilizan como “subrutinas” para realizar una tarea concreta. Una función que no devuelve un resultado, a veces se denomina “procedimiento

**Resultados de una función:** el valor devuelto (expresión) puede ser cualquier tipo de dato excepto una función o un array. Se puen devolver valores múltiples devolviendo un puntero a una estructura o a un array. El valor de retorno debe seguir las mismas reglas que se aplican a un operador de asignación

**Llamada a una función:** las funciones, para poder ser ejecutadas, han de ser llamadas o invocadas. Cualquier expresión puede contener una llamada a una función que redirigirá el control del programa a la función nombrada. La función que llama a otra función se denomina función llamadora y la función controlada se denomina función llamada

PROTOTIPOS DE LAS FUNCIONES

Los prototipos de una función contienen la misma cabecera de la función, con la diferencia de que los prototipos terminan con un punto y coma (;)

Un prototipo declara una función y proporciona una información suficiente al compilador para verificar que la función esta siendo llamada correctamente, con respecto al número y tipo de los parámetros y tipo de los parámetros y el tipo devuelto por la función. Es obligatorio poner un punto y coma al final del prototipo de la función con el objeto de convertirlo en una sentencia

Los prototipos se sitúan normalmente al principio de un programa, antes de la definición de la primera función “main ()”. El compilador utiliza los prototipos para validar que el número y los tipos de datos de los argumentos reales de la llamada a la función son los mimos que el número y tipo de argumentos formales en la función llamada

Cuando una entidad se declara, se proporciona un nombre y se listan sus características. Una definición proporciona un nombre de entidad y reserva espacio de memoria para esa entidad.

Una declaración de la función contiene solo la cabecera de la función y una vez declarada la función, la definición completa de la función debe existir en algún lugar del programa; por ejemplo, antes o después de “main”

**Int max (int, int); //declaración de max**

**Declaraciones una función**

* Antes de que una función pueda ser invocada, debe ser declarada
* Una declaración de una función contiene solo la cabecera de la función (llamado también prototipo)

**“tipo\_resultado nombre ( tipo1 param1, tipo2 param2,…);**

* Los nombres de los parámetros se pueden omitir

Double med (doublé, doublé);

Doublé med (double x1, double x2);

**//programa principal**

**Main ()**

**{**

**Int m, n;**

**Do**

**{**

**Cin>> m >>n;**

**Cout <<max (m, n) <<endl;**

**} while (m != n);**

**}**

**// devuelve el entero mayor**

**Int max (int x, int y) //definición de max**

**{**

**If (x<y) return y;**

**Else return x;**

**}**

* **prototipos con un número no especificado de parámetros:** un formato especial de prototipo es aquel que tiene un número no especificado de argumentos que se representa por puntos suspensivos (…)

**int muestras (int a, …);**

PARAMETOS DE UNA FUNCION

C++ proporciona dos métodos para pasar variables (parámetros) entre funciones. Una función puede utilizar “parámetros por valor y parámetros por referencia”, o puede no tener parámetros.

**Void circulo (int x, int y, int diámetro);**

Cuando se llama a “circulo” se deben pasar tres parámetros a esta función. En el punto de llamada cada parámetro puede ser una constante, una variable o una expresión

* **paso de parámetros por valor:** paso por valor significa que cuando C++ compila la función y el código que llama a la función, la función recibe una copia de los valores de los parámetros. Si se cambia el valor de un parámetro variable local, el cambio solo afecta a la función y no tiene efecto fuera de la función

**“nota:** el método por defecto de pasar parámetros es por valor, a menos que se pasen arrays. Los arrays se pasan siempre por dirección”

* **paso de parámetros por referencia:** cuando una función debe modificar el valor del parámetro pasado y devolver este valor modificado a la función llamadora, se ha de utilizar el método de paso de parámetro por referencia o dirección. En este método el compilador pasa la dirección de memoria del valor del parámetro a la función. Cuando se modifica el valor del parámetro (variable local), este valor queda almacenado en la misma dirección de memoria, por lo que al retornar a la función llamadora la dirección de la memoria donde se almaceno el parámetro contendrá el valor modificado. Para declarar una variable parámetro como paso por referencia, el símbolo “&” debe preceder el nombre de la variable

**void intercambio (int& m, int& n)**

**{**

**Int aux = m;**

**M = n;**

**N = aux;**

**}**

* **diferencia entre los parámetros por valor y por referencia**
  + los parámetros valor reciben copias de los valores de los argumentos que se les pasan
  + la asignación a parámetros valor de una función nunca cambian el valor del argumento original pasado a los parámetros
  + los parámetros referencia reciben la dirección de los argumentos pasados
  + en una función, las asignaciones a parámetros referencia cambian los valores de los argumentos originales
* **parámetros “const” de una función:** con el objeto de añadir seguridad adicional a las funciones, se puede añadir a una descripción de un parámetro el especificador “const”, que indica al compilador que solo es de lectura en el interior de la función. Si se intenta escribir en este parámetro se producirá un mensaje de error de compilación

**void falso (const int ítem, const int char& car)**

**{**

**Item = 123;**

**Car =`A`;**

**}**

ARGUMENTOS POR OMISION

Cuando una función tiene un cierto numero de parámetros, normalmente el mismo numero de argumentos deben indicarse cuando se llama a la función

Una característica poderosa de las funciones C++ es que en ellas pueden establecer valores por “omisión” o “ausencia” (por defecto) para los parámetros. Se pueden asignar argumentos por defecto a los parámetros de una función. Cuando se omite el argumento de un parámetro que es un argumento por defecto, se utilizan automáticamente este. La única restricción es que se deben incluir todas las variables desde la izquierda hasta el primer parámetro omitido

FUNCIONES EN LINEA (INLINE)

Las funciones en line sirven para aumentar la velocidad de su programa. Su uso es conveniente cuando la función se utiliza muchas veces en el programa y su código es pequeño. Existen una diferencia grande en el comportamiento de ambos tipos de funciones

Para una función en línea, el compilador inserta realmente el código para la función en el punto en que se llama la función. Esta acción hace que el programa se ejecute mas rápidamente, ya que no ha de ejecutar el código asociado con la llamada a la función

Sin embargo, cada instancia de la función en línea puede requerir tanta memoria como se requiera para contener la función completa. Por esta razón, el programa incrementa su tamaño, aun que es mucho mas rápido en su ejecución. Si se llama a una función en línea diez veces en un programa, el compilador inserta diez copias de la función en el programa. Si la función ocupa 1K, el tamaño de su programa se incrementa en 10K

**REGLA:** comience declarando las funciones en-línea como funciones ordinarias cuando desarrolle sus programas. Las funciones en línea son mas fáciles de depurar. Una vez que su programa este funcionando, inserte la palabra reservada “inline” donde sea necesario

* **creación de funciones en línea:** para crear una función en línea, se debe insertar la palabra reservada “inline” delante de una declaración normal y del cuerpo, y siturla en el archivo fuente antes de que sea llamada

**inline int sumar15 (int n) {return (n+15}**

AMBITO (ALCANCE)

El “ámbito” o alcance de una variable determina cuales son las funciones que reconocen ciertas variables. Si una función reconoce una variable, la variable es “visible” en esa función. El ámbito es la zona de un programa en el que es visible una variable. Normalmente la posición de la sentencia en el programa determina el ámbito

* **Ámbito del programa:** las variables que tienen ámbito de programa pueden ser referenciadas por cualquier función en el programa completo; tales variables se llaman variables globales

Una variable global es visible desde su punto de definición en el archivo fuente. Es decir, si se define una variable global, cualquier line del resto del programa, no importa cuantas funciones y líneas de código le sigan, podrá utilizar esa variable

**Consejo:** declare todas las variables en la parte superior de su programa. Aunque se pueden definir tales variables entre dos funciones, podría realizar cualquier cambio en su programa de modo mas rápido, si situa las variables globales al principio

* **Ámbito del archivo fuente:** una variable que se declara fuera de cualquier función y cuy declaración contiene la palabra reservada “static” tiene ámbito de archivo fuente. Las variables con este ámbito se pueden referenciar desde el punto del programa en que están declaradas hasta el final del archivo fuente
* **Ámbito de una función:** las variables declaradas dentro del cuerpo de la función se dice que son “locales” a la función. Las variables locales no se pueden utilizar fuera del ámbito de la funcon en que están definidas
* **Ámbito de bloque:** las variables locales declaradas dentro de una función tienen ámbito de bloque; no son visibles fuera del bloque
* **Variables locales:** existen en memoria solo cuando la función esta activa (es decir, mientras se ejecutan las sentencias de la función). Cuando la función no se esta ejecutando, sus variables locales no ocupan espacion en memoria ya que no existen

CLASES DE ALMACENAMIENTO

Los especificadores de clases (tipo) de almacenamiento permiten modificar el ámbito de una variable

* **Variables automáticas:** las variables que se declaran dentro de una función se dice que son automáticas (auto), significando que se les asigna espacio en memoria automáticamente l entrada de la función y se les libera el espacio tan pronto se sale de dicha función
* **Variables externas:** cuando una variable se declara externa, se indica al compilador que el espacio de la variable está definida en otro lugar
* **variables registros:** precediendo la declaración de una variable con la palabra reservada “register”, se sugiere al compilador que la variable se almacena en uno de los registros hardware del microprocesador. Una variable registro debe ser local a un función, nunca puede ser global al programa completo
* **variables estáticas:** las variables estáticas no se borran (no se pierde su valor) cuando la función termina y, en consecuencia, retienen sus valores entre llamadas a una función. Una variable “static” se inicializa solo una vez. Se declaran precediendo a la declaración de la variable con la palabra reservada “static”

**func\_uno ()**

**{**

**Int i;**

**Static int j = 25; //j, k variable estaticas**

**Static int k = 100;**

**}**

CONCEPTO Y USO DE FUNCIONES DE BIBLIOTECA

Todas las versiones del lenguaje C++ vienen con una biblioteca estándar de funciones en tiempo de ejecución que proporcionan soporte para operaciones utilizadas con más frecuencia

Las funciones estándar o predefinidas, como así se denominan las funciones pertenecientes a la biblioteca estándar, se dividen en grupos; todas las funciones que pertenecen al mismo grupo se declaran en el mismo archivo de cabecera

En los módulos de programa se pueden incluir líneas “#include” con los archivos de cabecera correspondientes en cualquier orden, y estas líneas pueden aparecer más de una vez

* **funciones de carácter:** el archivo de cabecera “<ctype.h>” define un grupo de funciones/macros de manipulación de caracteres. Todas las funciones devuelven un resultado de valor verdadero o falso

|  |  |  |
| --- | --- | --- |
| **Comprobación alfabética y de dígitos** | | |
| **Isalpha (c)** | Devuelve verdadero si “c” es una letra mayúscula o minúscula. | |
| **Islower (c)** | Devuelve verdadero si “c” es una letra minúscula y falso si es distinto de una minúscula | |
| **Isupper (c)** | Devuelve verdadero si “c” es una letra mayúscula y falso si es distinto de una mayúscula | |
| **Isdigit (c)** | Comprueba si “c” es un digito de 0 a 9 , devolviendo verdadero y falso en caso contrario | |
| **Isxdigit (c)** | Devuelve verdadero si “c” es cualquier digito hexadecimal y falso en cualquier otro caso | |
| **Isalnum (c)** | Devuelve un valor verdadero si “c” es un digito de 0 a 9 o un carácter alfabético | |
| **Funciones de prueba de caracteres especiales** | | |
| **Iscntrl (c)** | Devuelve verdadero si “c” es un carácter de control (código ASII 0 a 31) | |
| **Isgraph (c)** | Devuelve verdadero si “c” es un carácter imprimible excepto espacio | |
| **Isprint (c)** | Devuelve verdadero si “c” es un carácter imprimible, código ASCII 21 a 127 incluyendo espacio | |
| **Ispunct (c)** | Devuelve verdadero si “c” es cualquier carácter de puntuación (carácter imprimible) | |
| **Isspace (c)** | Devuelve verdadero si “c” es carácter un espacio, nueva line (\n), retorno, tabulación | |
| **Funciones de conversión de caracteres** | | |
| **Tolower (c)** | | Convierte el carácter “c” a minúscula |
| **Toupper (c)** | | Convierte el carácter “c” a minúscula si ya no lo es |
| **Strlen(c)** | | Muestra la longitud de una cadena |

* **funciones numéricas:** la mayoría de las funciones numéricas están en el archivo de cabecera “math.h”; las funciones “abs” y “labs” están definidas en “math.h” y “stdlib.h” y las rutinas “div” y “ldiv” en “stdlib.h”

|  |  |
| --- | --- |
| **Funciones matemáticas** | |
| **Ceil (x)** | Redondea el entero mas cercano |
| **Fabs (x)** | Devuelve el valor absoluto de x (un valor positivo) |
| **Floor (x)** | Redondea por defecto al entero mas próximo |
| **Fmod (x,y)** | Calcula el resto en coma flotante para la división “x/y”, de modo que “x=i\*y+f”, donde “i” es un entero, “f” tiene el mismo signo que “x” y el valor absoluto de “f” es menor que el valor de “y” |
| **Pow (x,y)** | Calcula “x” elevado a la potencia “y”. si “x” es menor que o igual a cero, “y” debe ser un entero. Si “x” es igual a cero, “y” no pude ser negativo |
| **Sqrt (x)** | Devuelve la raíz cuadrada de “x”, “x” deber ser mayor o igual a cero |
| **Funciones trigonométricas (math.h)** | |
| **Acos (x)** | Calcula el arco coseno del argumento “x”. el argumento debe estar entre -1 y 1 |
| **Asin (x)** | Calcula el arco seno del argumento “x”. el argumento “x” debe estar entre -1 y 1 |
| **Atan (x)** | Calcula el arco tangente del argumento “x” |
| **Atan2 (x,y)** | Calcula el arco tangente de x dividido por y |
| **Cos (x)** | Calcula el coseno del ángulo x”” (“x” se expresa en radianes) |
| **Sin (x)** | Calcula el seno del ángulo “x” (“x” se expresa en radianes) |
| **Tan (x)** | Devuelve la tangente del ángulo “x” (“x” se expresa en radianes) |
| **Funciones logarítmicas y exponenciales (math.h)** | |
| **Exp (x), expl (x)** | Calcula el exponencial “ex”, donde “e” es la base de logaritmos naturales de valor 2.718282 |
| **Log (x), logl (x)** | La función “log” calcula el logaritmo natural del argumento “x” y “logl (x)” calcula el algoritmo natural del argumento “x” de valor “long doublé” |
| **Log10 (x), log10l (x)** | Calcula el logaritmo decimal del argumento “x”, de valor real doublé en “log10 (x)” y de valor real “long doublé” en “log10l (x), “x” ha de ser positivo |
| **Funciones aleatorias (stdlib.h)** | |
| **Rand (void)** | La función “rand” genera un numero aleatorio. El numero calculado por “rand” varia en el rango de 0 a “RAND\_MAX (2E15-1)” |
| **Randomize (void)** | La macro “randomize” inicializa el generador de números aleatorios con una semilla aleatoria obtenida a partir de una llamada a la función “time”. Dado que esta macro llama a la función “time”, el archivo de cabecera “time.h” se incluirá en el programa |
| **Srand (semilla)** | La función “srand” inicializa el generador de números aleatorios. Se utiliza para fijar el punto de comienzo para la generación de series de números aleatorios; este valor se denomina “semilla” |
| **Random (num)** | La macro “random” genera un numero aleatorio dentro de un rango especificado (0 y limite superior especificado por el argumento “num”) |

* **funciones de fecha y hora:** el archivo de cabecera “time.h” define estructuras, macros y funciones para manipulación de fecha y horas. Las funciones “time”, “clock”, “\_strdate” y “\_strtime”, devuelven la hora actual como el numero de segundos transcurridos

|  |  |
| --- | --- |
| **Clock (void)** | La función “clock” determina el tiempo de ejecución transcurrido desde el principio de la ejecución del programa; es decir determina el numero de segundos |
| **Time (hora)** | La función “time” obtiene la hora actual |
| **Localtime (hora)** | Convierte la fecha y hora en una estructura de tipo “tm”. Su prototipo es  *Struct tm \*localtime (const time\_T \*tptr);* |
| **Mktime (t)** | Convierte la hora a un formato de calendario. Toma la información de la hora local contenida en la estructura “\*tptr” y determina los valores de todos los miembros en el formato de tiempo del calendario |

* **funciones de utilidad:** se incluye una serie de funciones de utilidad que se encuentran en el archivo de cabecera “stdlib.h” y que se listan a continuación

|  |  |
| --- | --- |
| **Abs (n), labs (n)** | Devuelven el valor absoutl de n; “*int abs (int n);”* |
| **Div (num, denom)** | Calcula el cociente y el resto de “num”, divido por “denom” y almacena el resultado en “quot” y “rem”, miembros de “int” de la estructura “div\_t”  *Typedef struct {int quot; int rem;}div\_T* |
| **Ldiv (num, denom)** | Calcula el cociente y resto de “num” dividido por “denom” y almacena los resultados de “quot” y “rem”, miembros de “long” de la estructura “ldiv\_t” |

VISIBILIDAD DE UNA FUNCION

El ámbito de una objeto es su visibilidad desde otras partes del programa y la duración de un objeto es su tiempo de vida, lo que implica no solo cuanto tiempo existe la variable, sino cuando se crea y cuando se hace disponible, se puede decir que un objeto definido dentro de una función tiene ámbito local (alcance local), o si se define fuera de cualquier función, se dice que tiene un ámbito global

Existen dos tipos de clases de almacenamiento en C++: “auto” y “static”. Una variable “auto” es aquella que tiene una “duración automática”. No existe cuando el programa comienza la ejecución, se crea en algún punto durante la ejecución y desaparece en algún punto antes de que el programa termine la ejecución. Una variable “static” es aquella que tiene una “duración fija”. El espacio para el objeto se establece en tiempo de compilación: existe en tiempo de ejecución y se elimina solo cuando el programa desaparece de memoria en tiempo de ejecución

* **Variables locales frente a variables globales:** una variable local esta definida solamente dentro del bloque o cuerpo de la función y no tiene significado (vida) fuera de la función respectiva. Por consiguiente, si una función define una variable como local, el ámbito de la variable esta protegido. La variable no se puede utilizar, cambiar o borrar desde cualquier otra función sin una programación específica mediante el paso de valores (parámetros)
* **Variables estáticas y automáticas:** los valores asignados a las variables locales de una función se destruyen cuando se termina la ejecución de la función y no se puede recuperar su valor para ejecuciones posteriores de la función. Las variables locales se denominan “variables automáticas”

Las variables estáticas mantienen su valor después que una función se ha terminado. Una variable de una función declarada como estática, mantiene un valor a través de ejecuciones posteriores de la misma función. Haciendo una variable local estática, su valor se retiene de una llamada a la siguiente de la función en que está definida

COMPILACION SEPARADA

Los programas grandes son mas fáciles de gestionar si se dividen en varios archivos fuentes, también llamados “módulos”, cada uno de los cuales puede contener una o mas funciones. Estos módulos se compilan y enlazan por separado posteriormente con un “enlazador”, o bien con la herramienta correspondiente del entorno de programación. Cuando se divide un programa grande en pequeños, los únicos archivos que se recompilan son son los que se han modificado. El tiempo de compilación se reduce, dado que pequeños archivos fuente se compilan mas rápido que los grandes. Los archivos grandes son difíciles de mantener y editar, ya que su impresión es un proceso lento que utilizara cantidades excesivas de papel

Cuando se tiene mas de un archivo fuente, se puede referenciar una función en un archivo fuente desde una función de otro archivo fuente. Al contrario que las variables, las funciones son externas por defecto.

**// MODULO1. Cpp**

**#include <iostream.h>**

**Main ()**

**{**

**Void f (int i), g (void);**

**Extern int n; //declaracion de n (no definicion)**

**F (8);**

**N++;**

**G ();**

**Cout<<”fin de programa \n”;**

**}**

**// MODULO2. Cpp**

**#include <iostream.h>**

**Int n = 100; //delcaracion de n (también declaración)**

**Static int m = 7;**

**Void f (int i)**

**{**

**N += i+m;**

**}**

VARIABLES REGISTRO (register)

Una variable registro es similar a una variable local, pero en lugar de ser almacenada en la pila, se almacena directamente en un registro del procesador. Dado que el número de registro es limitado y además están limitados en tamaño, el número de variables registro que un programa puede crear simultáneamente es muy restringido

La ventaja de las variables registro es su mayor rapidez de manipulación. Esto se debe a que las operaciones sobre valores situados en los registros son normalmente mas rápidas que cuando se realizan sobre valores almacenados en memoria. La variable registro pueden ayudar a optimizar el rendimiento de un programa proporcionado acceso directo de la CPU a los valores claves del programa

**Void usoregistro (void)**

**{**

**Register int k;**

**Cout<<”\n contar con una variable registro \n”;**

**For (k=1; K<=100; k++)**

**Cout<<stew (8) << dec << k;**

**}**

SOBRECARGA DE FUNCIONES (POLIMORFISMO)

la sobrecarga de funciones permite escribir y utilizar múltiples funciones con el mismo nombre, pero con diferente lista de argumentos. La lista de argumentos es diferente si tiene un argumento con un tipo de dato distinto, si tiene un número diferente de argumentos, o ambos. La lista de argumento se suele denominar “signatura de la función

RECURSIVIDAD

Una función recursiva es una función que se llama así misma directa o indirectamente. La recursividad o recursión directa es el proceso por el que una función se llama así misma desde el propio cuerpo de la función. La recursividad o recursión directa es el proceso por el que una función se llama así misma desde el propio cuerpo de la función. La recursividad o recursión indirecta implica más de una función

**UNIDAD VI: recursividad**

FUNCIONES RECURSIVAS

Una función recursiva es una función que se invoca a si misma. En recursión directa el código de la función “F” contiene una sentencia que invoca a “F”, mientras que en recursión indirecta la función “F” invoca a una función “G” que invoca a su vez a la función “H”, y así sucesivamente, hasta que se invoca de nuevo la función “F”

* **PROBLEMA:** escribir una función Fibonacci, de modo que calcule el valor del elemento de acuerdo a la posición ocupada en la serie

**//función de Fibonacci: fibo.cpp**

**#include<iostream>**

**Usinged long Fibonacci (unsigned long);**

**Main ()**

**{**

**Unsigned long resultado, num;**

**Cout<<”introduzca un entero: “;**

**Cin>>num;**

**Resultado = Fibonacci (num);**

**Cout<<”el valor de Fibonacci(“<<num<<”)=”resultado”<<endl;**

**Return 0;**

**}**

**//definición recursiva de la función de Fibonacci**

**Usinged long Fibonacci (usigned long n)**

**{**

**If (n==0||n==1)**

**Return n;**

**Else**

**Return Fibonacci (n-1)+Fibonacci(n-2);**

**}**

* **Funciones mutuamente recursivas:** la recursividad indirecta se produce cuando una función llama a otra, que eventualmente terminara llamando de nuevo a la primera función

**//listado ALFABETO.cpp**

**#include <iostream>**

**#include <stdio.h>**

**Void A (int c);**

**Void B (int c);**

**Main ()**

**{**

**A(`z`);**

**Cout<<endl;**

**Return 0;**

**}**

**Void A (int c)**

**{**

**If (c>`A`)**

**B(c);**

**Putchar (c);**

**}**

**Void B (int c)**

**{**

**A(--c);**

**}**

El programa principal llama a la función recursiva “A ()” con el argumento `z`. La función “A” examina su parámetro `c`. Si `c` está en orden alfabético después que “A”, la función llama a “B()”, que inmediatamente llama a “A()”, pasándole un parámetro predecesor de `c`. Esta acción que hace que “A()” vuelva a examinar a `c`, y nuevamente una llamada a “B()”, hasta que c sea igual a `a`. en este momento, la recursión termina ejecutando “putchar ()” veintiséis veces y visualizando el alfabeto, carácter a carácter

* **Condición de terminación de la recursión:** cuando se implementa una función recursiva será preciso considerar una condición de terminación, ya que en caso contrario la función continuaría indefinidamente llamándose a si misma y llegaría un momento en que la memoria se podría agotar

**Long factorial (long n)**

**{**

**If (n==1)**

**Return 1;**

**Else**

**Return n\*factorial (n-1);**

**}**

RECURSION FRENTE A ITERACION

Tanto la iteración como la recursión se basan en una estructura de control: *la iteración utiliza una estructura de repetición y una recursión utiliza una estructura de selección.*

La iteración y recursión implican cada una un “test” de terminación (condición de salida). La iteración termina cuando la condición del bucle no se cumple, mientras que la recursión termina cuando se reconoce un caso base o la condición de salida se alcanza.

La recursión tiene muchas desventajas. Se invoca repetidamente al mecanismo de recursividad y, en consecuencia, se necesita tiempo suplementario para realizar las llamadas a funciones

***“consejo de programación:*** *se ha de evitar utilizar recursividad en situaciones de rendimiento critico o exigencia de altas prestaciones en tiempo y memoria, ya que las llamadas recursivas empelan tiempo y consumen memoria adicional****”***

RECURSION INFINITA

La recursión infinita significa que cada llamada recursiva produce otra llamada recursiva y esta a su vez otra llamada recursiva y así para siempre.

El flujo de control de una función recursiva requiere tres condiciones para una terminación normal:

* Un test para detener (o continuar) la recursión
* Una llamada recursiva (para continuar la recursión)
* Un caso final para terminar la recursión

ORDENACION RAPIDA (quicksort)

La idea del algoritmo es simple, se basa en la división en particiones de la lista a ordenar. El método es, posiblemente, el más pequeño de código, mas rápido, más elegante y más interesante y eficiente de los algoritmos conocidos de ordenación.

El método se basa en dividir los “n” elementos de la lista a ordenar en tres partes o particiones: una partición “izquierda” , un partición “central”, que solo contenga un elemento denominado “pivote” o elemento de partición, y una partición “derecha”. La partición o división se hace de tal forma que todos los elementos de la primera sub-lista (partición izquierda) son menores que todos los elementos de la segundo sub-lista (partición derecha). Las dos sub-lista se ordenan entonces independientemente.

La lista se divide se particiones eligiendo uno de los elementos de la lista y se utiliza como pivote. Si se elige una lista cualquiera con elementos en orden aleatorio, se puede elegir cualquier elemento de la lista como pivote. Si la lista tiene algún orden parcial, que se conoce, el pivote se debe elegir de modo que se divida la lista exactamente por la mitad, de acuerdo al tamaño relativo de las claves.

Una vez que el pivote ha sido elegido, se utiliza para ordenar el resto de la lista en dos sub-lista: una tiene todas las claves menores que el pivote y la otra en el que todos los elementos (claves) son mayores o iguales que el pivote. Etas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llma sucesivamente al propio algoritmo “quickshort”. La lista final ordenada se consigue concatenando la primera sub-lista, el pivote y la segunda lista, en ese orden, en una única lista. El algoritmo “quicksort” requiere una estrategia de partición y la selección idónea del pivote. Las etapas fundamentales del algoritmo dependen del pivote elegido aunque la estrategia de partición suele ser similar. Intercambio será una listaDe estLa primera etapa en el algoritmo de partición es obtener el elemento pivote; una vez que se ha seleccionado se ha de buscar el sistema para situar en la sub-lista izquierda todos los elementos menores o iguales que el pivote y en la sub-lista derecha todos los elementos mayores que el pivote.se intercambia el pivote con el último elemento. El resultado de este intercambio será una nueva lista

**Por ejemplo**

**Lista inicial:**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **2** | **96** | **18** | **38** | **12** | **45** | **10** | **55** | **81** | **43** | **39** |

**Pivote elegido: 38**

**Sub-lista izquierda menores que 38**

|  |  |  |  |
| --- | --- | --- | --- |
| **2** | **18** | **12** | **10** |

**Sub-lista derecha mayor que 38**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **96** | **45** | **55** | **81** | **43** | **39** |

**Lista ordenada**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **2** | **18** | **12** | **10** | **38** | **96** | **45** | **55** | **81** | **43** | **39** |

**UNIDAD VII: arrays (listas y tablas)**

INTRODUCCION A LAS ESTRUCTURAS DE DATOS

Una estructura de datos es una colección de datos que pueden ser caracterizados por su organización y las operaciones que se definen en ella

Los tipos de datos mas frecuentes utilizados en los diferentes lenguajes de programación son; los tipos de datos “simples” o “primitivos” significan que no están compuestos de otra estructuras de datos; las mas frecuentes y utilizados por casi todos los lenguajes son: enteros, reales y caracteres, siendo los tipos lógicos, sub-rangos y enumerativos propios de lenguajes estructurados. Los tipos de datos compuestos están construidos basados en tipos de datos primitivos

Los tipos de datos simples pueden ser organizados en diferentes estructuras de datos: “estáticas” y “dinámicas”. Las **estructuras de datos estáticas** son aquellas en las que el tamaño ocupado en memoria se define antes de que el programa se ejecute y no puede modificarse dicho tamaño durante la ejecución del programa. Las  **estructuras de datos dinámicas** no tienen las limitaciones o restricciones en el tamaño de memoria ocupada que son propias de las estructuras estáticas. Mediante el uso de un tipo de datos especifico, denominado “puntero”, es posible construir estructuras de datos dinámicas que son soportadas por la mayoría de los lenguajes ofrecen soluciones eficaces y efectivas en la solución de problemas complejos

Una característica importante que diferencia a los tipos de datos es la siguiente: los tipos de datos simples tiene como característica común que cada variable representa a un elemento; los tipos de datos estructurados tienen como característica común que un identificador (nombre) puede representar múltiples datos individuales, pudiendo cada uno de estos ser referenciado independientemente

ARRAYS

Un array (lista o tabla) es una secuencia de objetos del mismo tipo. Los objetos se llaman elementos del array y se numeran consecutivamente 1,2,3… estos numero se denominan “valores índices” o “sub-índices del array”. El termino subíndice se utiliza ya que se especifica igual que en matemáticas como una secuencia tal como a0 .a1 ,a2 … estos números localizan la posición del elemento dentro del array, proporcionando acceso directo al array

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| a | 25.1 | 34.2 | 5.25 | 7.45 | 6.09 |
|  | 0 | 1 | 2 | 3 | 4 |

* **declaración de un array** un array se declara de modo similar a otros tipos de datos, excepto que se debe indicar al compilador el tamaño o longitud del array. Para indicar al compilador el tamaño o longitud del array se debe hacer seguir el nombre, el tamaño encerrado entre corchetes

**int números [10];**

esta declaración hace que el compilador reserve espacio suficiente para contener diez valores enteros. En C++ los enteros ocupan, normalmente, 2 bytes, de modo que un array de diez enteros ocupa 20 bytes de memoria

* **subíndices de un array:** el índice de un array se denomina, con frecuencia, subíndice del array. El método de numeración del elemento i-esimo con el índice o subíndice i-l se denomina “indexación basada en cero”

**numeros0  equivale números [0]**

* **almacenamiento en memoria de los arrays:** los arrays de caracteres funcionan de igual forma que los arrays numéricos, partiendo de la base de que cada carácter ocupa normalmente un byte.
* **El tamaño de los arrays (sizeof):** la función “sizeof ()” devuelve el numero de bytes necesarios para contener su argumento. Si se usa sizeof () para solicitar el tamaño de un array, esta función devuelve el numero de bytes reservados para el array completo

**N=sizeof (edades []);**

INICIALIZACION (INICIACION) DE UN ARRAY

Se debe asignar valores a los elementos del array antes de utilizarlos, tal como se asignan valores a variables.

Cuando se inicializa un array, el tamaño del array se puede determinar automáticamente por las constantes de inicialización. Estas constantes se separan por comas y se encierran entre llaves “{}”

**Int numero [6**

**]={10, 20, 30, 40, 50, 60}**

El método de inicializar arrays mediante valores constantes después de su definición es adecuado cuando el numero de elementos del array es pequeño

Se pueden asignar valores a un array utilizando un bucle “for o while/do-while”;

**Int cuenta [10]={10,9,8,7,6,5,4,3,2,1,0};**

**For (i=9;i>=1;i--)**

**Cout<<”\n cuenta descendente”<<i<<”=”<<cuenta[i];**

También se pueden asignar constantes simbólicas como valores numéricos, de modo que las sentencias siguientes son validas

**Const int ENE=31, FEB=28, MAR=31, ABR=30, MAY=31, JUN=30, JUL=31, AGO=31, SEP=30, OCT=31, NOV=30,DIC=31;**

**…**

**INT MESES [12]= {ENE, FEB. MAR, ABR. MAY, JUN, JUL, AGO, SEP, OCT, NOV, DIC}**

**NOTA:** C++puede dejar los corchetes vacios, solo cuando se asignan valores al array, tal como **“int cuenta []={15,25,-45,0.25}”**

ARRAYS DE CARACTERES Y CADENAS DE TEXTO

Es importante comprender la diferencia entre un array de caracteres y una cadena de caracteres. Las cadenas contienen un carácter nulo al final de array de caracteres. Las cadenas se señalan incluyen un carácter al final de la cadena: el carácter nulo (\0), cuyo valor en el código ASCII es 0. El medio mas fácil de inicializar un array de caracteres es hacer la inicialización de la declaración. El compilador inserta automáticamente un carácter nulo al final de la cadena, de modo que la secuencia real seria:

**Char cadena [7]=”ABCDEF”;**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Cadena | A | B | C | D | E | F | \0 |

Para copiar una constante cadena o copiar una variable de cadena a otra variable de cadena se debe utilizar la función de la biblioteca estándar “strcpy(), que permite copiar una constante de cadena en una cadena

ARRAYS MULTIDIMENSIONALES

Los arrays multidimensionales son aquellos que tienen más de una dimensión y, en consecuencia, más de un índice. Los arrays más usuales son los de dos dimensiones, conocidos también como tablas o matrices

**Nota:** C++ requiere que cada dimensión este encerrada entre corchetes la sentencia

**Int variable [4] [30]**

* **Inicialización de arrays multidimensionales:** los arrays multidimensionales se pueden inicializar, al igual que los de una dimensión, cuando se declaran. La inicialización consta de una lista de constantes separadas por comas y encerradas entre llaves

**NOTA:** los arrays multidimensionales (a menos que sean globales) no se inicializan a valores específicos a menos que se les asignen valores en el momento de la declaración o en el programa. Si se inicializan uno o mas elementos pero no todos. C++ rellena el resto con ceros o valores nulos (\0). Si se desea inicializar a cero un array multidimensional, utilice la sentencia

**Flota varialbe [1] [3] = {0,0}**

* **Acceso a los elementos de arrays bidimensionales:** se puede acceder a los elementos de arrays bidimensionales de igual forma que a los elementos de un array unidimensional. La diferencia reside en que los elementos bidimensionales deben especificarse los índices de la fila y la columna

**<Nombre array>[índice fila] [índice columna]=valor elemento (inserción de elementos)**

**<Variable>=<nombre array>[índice fila] [índice columna] (extracción de elementos)**

* **Lectura y escritura de elementos de arrays bidimensionales:** la sentencia “cin” y “cout” se utilizan para extraer elementos de arrays
* **Acceso a elementos mediante bucles:** se puede acceder a los elementos de arrays bidimensionales mediante bucles anidados

**For (int inicefila=0;indicefila<numfilas;++indiefila);**

**Procesar element [inidefila] [indicecolumna]**

* **Arrays de más de dos dimensiones:** C++ proporciona la posibilidad de almacenar varias dimensiones, aunque raramente los datos el mundo real requieren mas de dos o tres dimensiones. Un array tridimensional se puede considerar como un conjunto de arrays bidimensionales combinados juntos para formar, en profundidad, una tercera dimensión. El cubo se construye con filas (dimensión vertical), columna (dimensión horizontal) y planos (dimensión en profundidad)

UTILIZACION DE ARRAYS COMO PARAMETROS

En C++todos los arrays se pasan por referencia (dirección). Esto significa que cuando se llama a una función y se utiliza un array como parámetro, se debe tener cuidado de no modificar los arrays en una funion llamada. C++ trata automáticamente la llamada a la función como si hubiera situado el operador de dirección “&” delante del nombre del array

Dada las declaraciones

**Const MAX 100**

**Doublé datos [MAX];**

Se puede declarar una función que acepte un array de valores doublé como parámetros. Se puede prototipar una función “Suma-Datos”, de modo similar a:

**Doublé Suma-Datos (doublé datos [], int n)**

A la función “Suma-Datos” se pueden entonces pasar argumentos de tipo array junto con un entero “n”, que informa a la función sobre cuantos valores contiene el array. Por ejemplo, esta sentencia visualiza la suma de valores de los datos del array:

**Cout<<”suma de “<<Suma-Datos (datos, MAX);**

La función “Suma-Dato” no es difícil de escribir. Un simple bucle “for” suma los elementos del array y una sentencia “return” devuelve el resultado de nuevo al llamador

**Doublé Suma-Datos (doublé datos [], int n)**

**{**

**Doublé Suma =0;**

**While (n>0)**

**{**

**Suma+=datos [n];**

**Return Suma;**

**}**

**}**

* **Precauciones:** cuando se utiliza una variable array como argumento, la función receptora puede no conocer cuantos elementos existen en el array. Sin su conocimiento una función no puede utilizar el array. Aunque la variable array puede apuntar al comienzo del array, no proporciona ninguna indicación de donde termina el array

todas las cadenas terminadas en nulo utilizan el primer método. Una segunda alternativa es pasar el numero de elementos del array siempre que se pasa el array como un argumento. El array y el numero de elementos se convierten entonces en una pareja de argumentos que se asocian con la función llamada.

El segundo argumento, “No-Elementos”, es un valor entero que indica a la función cuantos elementos se procesaran en el array “Array-Enteros”. Este método suele ser el utilizado para arrays de elementos que no son caracteres

**#include<iostream>**

**Int SumaEntero (int \*ArrayEnteros, int NoElementos)**

**{**

**Int i, total=0;**

**For (i=0; i<NoElementos; i++)**

**{**

**Total +=ArrayEnteros [i];**

**Return Total;**

**}**

**}**

**Main ()**

**{**

**Int items [10];**

**Int Total, I;**

**Cout<<”intoduzca 10 numeros, seguidos por return”<<endl;**

**For (i=0;i<10;i++)**

**{**

**Cin>>ítems [i];**

**Total=SumaEntero(ítems, 10);**

**Cout<<Total;**

**Return 0;**

**}**

* **Paso de cadenas como parámetros:** la técnica de pasar arrays como parámetros se utiliza para pasar cadenas de caracteres a funciones. Las cadenas terminadas en nulo utilizan el primer método dado anteriormente para controlar el tamaño de un array. Las cadenas son arrays de caracteres. Cuando una cadena se pasa a una función, tal como “strlen ()” la función conoce que se ha almacenado el final del array cuando ve un valor de 0 en un elemento del array

BUSQUEDA DE ARREGLOS

* **Búsqueda secuencial:** el método mas sencillo de búsqueda consiste en explorar secuencialmente los elementos de un arreglo comparando cada uno con el criterio de búsqueda hasta que este se encuentra o hasta que el arreglo se lee por completo. La búsqueda secuencial no requiere de ningún requisito y, por consiguiente, no se necesita que el arreglo este ordenado. El recorrido del vector se realiza normalmente con estructuras repetitivas.

**Función busqueda\_sec(E num:vector; E buscado:entero): entero**

**Variables:**

**i: entero**

**encontrado: lógico**

**inicio**

**encontrado<-FALSO**

**i<-1**

**Mientras ((i<=MAX) Y encontrado=FALSO) hacer**

**si (buscado=num[i]) entonces**

**encontrado<-VERDADERO**

**sino**

**i<-i+1**

**fin\_si**

**fin\_mientras**

**si (encontrado=VERDADERO) entonces**

**busqueda\_sec<-i**

**sino**

**busqueda\_sec<-0**

**fin\_si**

**fin**

* **Búsqueda binaria:** la búsqueda binaria parte de la premisa de un vector ordenado y utiliza el método “divide y vencerás” para encontrar el valor buscado. Básicamente, se examina primero el elemento central del arreglo, si este es el buscado, entonces finaliza allí; de lo contrario se determina si el elemento buscado esta en la primera o la segunda mitad del arreglo y a continuación se repite el proceso, utilizando el elemento central del sub-arreglo

**Función busqueda\_bin(E num: vector, E ocup: entero, E buscado: entero): entero**

**Variables:**

**alto,bajo,central: entero**

**encontrado: lógico**

**inicio**

**bajo<-1**

**alto<-ocup**

**encontrado<-FALSO**

**mientras ((encontrado=FALSO) Y (bajo<=alto)) hacer**

**central<-(bajo+alto) div 2;**

**si (buscado=num[central]) entonces**

**encontrado<-VERDADERO**

**sino**

**si (buscado<num[central]) entonces**

**alto<-central-1**

**sino**

**bajo<-central+1**

**fin\_si**

**fin\_si**

**fin\_mientras**

**si (encontrado=VERDADERO) entonces**

**busqueda\_bin<-central**

**sino**

**busqueda\_bin<-0**

**fin\_si**

**fin**

ORDENACION DE LISTAS

La ordenación o clasificación es el procedimiento mediante el cual se disponen los elementos del array en un orden especificado, tal como orden alfabético u orden numérico. El orden de clasificación u ordenación puede ser ascendente o descendente

Existen numerosos algoritmos de ordenación de arrays: inserción, burbuja, selección, rápido (quick sort), fusión (merge), montículo (heap), Shell, etc.

* **Algoritmo de la burbuja:** la ordenación por burbuja es uno de los métodos mas fáciles de ordenación. El método de ordenación es muy simple. Se compara cada elemento del array con el siguiente, si no están en el orden correcto, se intercambian entre si sus valores.

La ordenación de arrays requiere siempre un intercambio de valores, cuando estos no se encuentran en el orden correcto; armar un programa que ordene una lista de números reales y a continuación los imprima

**procedimiento cambio(E/S x:entero, E/S y:entero)**

**variables:**

**aux:entero**

**inicio**

**auxx**

**xy**

**yaux**

**fin**

**procedimiento burbuja(E/S a:vector,E ocup:entero)**

**variables:**

**j:entero**

**bandera: lógico**

**inicio**

**banderaVERDADERO**

**mientras bandera hacer**

**banderaFALSO**

**para j desde 1 hasta ocup-1 hacer**

**si a[j]>a[j+1] entonces**

**cambio(a[j],a[j+1])**

**banderaVERDADERO**

**fin\_si**

**fin\_para**

**fin\_mientras**

**fin**

* **Algoritmo por Selección:** se basa en buscar el elemento menor del arreglo y colocarlo en primera posición. Luego se busca el segundo elemento mas pequeño y se coloca en la segunda posición, y así sucesivamente

Los pasos sucesivos a dar son:

* + Seleccionar el elemento menor del arreglo de n elementos
  + Intercambiar dicho elemento con el primero
  + Repetir estas operaciones con los n-1 elementos restantes, seleccionando el segundo elemento; continuar con los n-2 elementos restantes hasta que solo quede el mayot

**procedimiento seleccion(E/S a:vector,E ocup:entero)**

**variables:**

**i,j:entero**

**inicio**

**para i desde 1 hasta ocup-1 hacer**

**para j desde i+1 hasta ocup hacer**

**si a[i]>a[j] entonces**

**cambio(a[i],a[j])**

**fin\_si**

**fin\_para**

**fin\_para**

**fin**

* **Algoritmo por inserción:** ordena un vector insertando cada “elemento [i]” entre los i-1 anteriores que ya están ordenados. Para realizar esto comienza a partir del segundo elemento, suponiendo que el primero ya esta ordenado. Si los 2 primeros elementos están desordenados, los intercambia. Luego, toma el tercer elemento y busca su posición correcta con respecto a los dos primeros. En general, para el elemento i, busca su posición con respecto a los i-1 elementos anteriores y de ser necesario lo inserta adecuadamente

**procedimento insercion(E/S a:vector,E ocup:entero)**

**variables:**

**i,j,aux:entero**

**inicio**

**para i desde 2 hasta ocup hacer**

**auxa[i]**

**ji-1**

**mientras ((j>=1) Y (a[j]>aux)) hacer**

**a[j+1]a[j];**

**jj-1**

**fin\_mientras**

**a[j+1]aux**

**fin\_para**

**fin**

* **Algoritmo Shell:** también llamado inserción con incrementos decrecientes es una mejora del método de inserción. En este método se realizan comparaciones por saltos constantes (mayores a 1) con lo que se consigue una ordenación mas rápida. La ordenación Shell se basa en tomar como salto N/2 (N elementos del arreglo) y luego el salto se va reduciendo a la mitad en cada repetición hasta que vale 1

**procedimiento shell(E/S a:vector,E ocup:entero)**

**variables:**

**i,j,k,salto:entero**

**inicio**

**saltoocup div 2**

**mientras salto > 0 hacer**

**para i desde (salto+1) hasta ocup hacer**

**ji-salto**

**mientras j>0 hacer**

**kj+salto**

**si a[j]<=a[k] entonces**

**j0**

**sino**

**cambio(a[j],a[k])**

**fin\_si**

**jj-salto**

**fin\_mientras**

**fin\_para**

**saltosalto div 2**

**fin\_mientras**

**fin**

* **Algoritmo Quicksort:** permite ordenar una lista basándose en el hecho de que es mas fácil ordenar 2 listas pequeñas que una lista grande. El método divide al arreglo en 2 sub-lista, una con todos los valores menores o iguales a un cierto valor especifico y otra con todos los valores mayores que ese valor. El valor elegido puede ser cualquier valor arbitrario del vector. A este valor se lo denomina pivote

**procedimiento rapido(E/S a:vector,E izq:integer,E der:integer)**

**variables:**

**i,j,pivote:entero**

**inicio**

**iizq**

**jder**

**pivotea[(izq+der) div 2]**

**mientras i<= j hacer**

**mientras a[i]<pivote hacer**

**ii+1;**

**fin\_mientras**

**mientras a[j]>pivote hacer**

**jj-1**

**fin\_mientras**

**si i<= j entonces**

**cambio(a[i],a[j]);**

**ii+1;**

**jj-1;**

**fin\_si**

**fin\_mientras**

**si izq < j entonces**

**rapido(a,izq,j)**

**fin\_si**

**si i < der entonces**

**rapido(a,i,der)**

**fin\_si**

**fin**

BUSQUEDA EN LISTAS

La búsqueda de un elemento dado en un array es una aplicación muy usual en el desarrollo de programas en C++. Dos algoritmos típicos que realizan esta tarea son la búsqueda secuencial o en serie y la búsqueda binaria o dicotómica. La búsqueda secuencial es el método utilizado para listas no ordenadas, mientras que la búsqueda binaria se utiliza en arrays que ya están ordenados

ESTRUCTURAS Y UNIONES

* **Estructuras:** los arrays son estructuras de datos que contienen un numero determinado de elementos y todos los elementos han de ser del mismo tipo de datos. Esta característica supone una gran limitación cuando se requieren grupos de elementos con tipos diferentes de datos cada uno.

Los componentes individuales de una estructura se llaman miembros. Cada miembro (elemento) de una estructura puede contener datos de un tipo diferente de otros miembros

Una estructura es una colección de uno o más tipos de elementos denominados miembros, cada uno de los cuales puede ser un tipo de dato diferente

1. **Declaración de una estructura:** una estructura es un tipo de dato definido por el usuario, que se debe declarar antes de que se pueda utilizar

**Struct <nombre de la estructura>**

**{**

**<tipo de dato miebro1> <nombre miembro1>;**

**}**

1. **Definición de variable de estructuras:** al igual que a los tipos de datos enumerados, a una estructura se accede utilizando una variable o variables que se deben definir después de la declaración de la estructura. Del mismo modo que sucede en otra situación, en C++ existen dos conceptos similares a considerar, “declaración y definición”. Una declaración especifica simplemente el nombre y el formato de la estructura de datos, pero no reserva almacenamiento en memoria. Por consiguiente, cada definición de variable para una estructura dada crea un área en memoria en donde los datos se almacenan de acuerdo al formato estructurado declarado. Las variables de estructuras se pueden definir de dos formas: (1) listándolas inmediatamente después de la llave de cierre de la declaracion de la estructura, o (2) listando el nombre de la estructura seguida por las variable correspondientes en cualquier lugar del programa antes de utilizarlas

**//primera forma**

**Estruct coleciones\_CD**

**{**

**Char titulo [30];**

**Char artista [25];**

**Int num\_canciones;**

**Float precio;**

**Char fecha\_compra [8];**

**} cd1, d2, cd3;**

**//segunda forma**

**Colecciones\_CD cd1, cd2, cd3;**

1. **Uso de estructuras en asignaciones:** como una estructura es un tipo de dato similar a un “int” o “char”, se puede asignar una estructura a otra

**Libro3=Libro1;**

**Libro4=Libro1;**

**Libro5=Libro1;**

1. **Inicialización de una declaración de estructuras:** se puede inicializar una estructura de dos formas. Se puede inicializar una estructura dentro de la sección de código de su programa, o bien se puede inicializar la estructura como parte de la declaración. Cuando se inicializa una estructura como parte de la declaración, se especifican los valores iníciales, entre llaves, después de la declaración

**Struct <nombre variable estructura> =**

**{valor miembro1,**

**Valor miembro2,**

**};**

**Struct info\_libro**

**{**

**Char titulo [60];**

**Char autor [30];**

**Char editorial [30];**

**Int anyo;**

**} libro1={“C++ a su alcance, “luis joyanes”, “mcgrawhill” , 1994};**

1. **El tamaño de una estructura:** uso del operador “sizeof” para determinar el tamaño

**#include <iostream>**

**//declarar una estructura persona**

**Estruct Persona**

**{**

**Char nombre [30];**

**Int edad;**

**Float altura;**

**Float peso;**

**};**

**Void main ()**

**{**

**Persona persona;**

**Cout<<”sizeof (Persona): “<<sizeof (Persona);**

**}**

ACCESO A ESTRUCTURAS

Cando se accede a una estructura, o bien se almacena información en la estructura o se recupera la información e la estructura. Se puede acceder a los miembros de una estructura de una de estas dos formas: (1) utilizando el operador punto (.), o bien (2) utilizando el operador puntero (🡪)

1. **Almacenamiento de información en estructuras:** se puede almacenar información en una estructura mediante inicialización, asignación directa o lectura del teclado.

**Acceso a una estructura de datos mediante el operador punto**

La asignación de datos a estructuras se puede hacer mediante el operador punto

**<nombre variable estructura> . <nombre miembro> = datos;**

El operador punto proporciona el camino directo al miembro correspondiente. Los datos que se almacenan en un miembro dado deben ser del mismo tipo que el tipo declarado para ese miembro

**Acceso a una estructura de datos mediante el operador puntero**

El operador puntero “ 🡪 “ sirve para acceder a los datos de la estructura. Para utilizar este operador se debe definir primero una variable puntero para apuntar a la estructura.

**<puntero estructura> 🡪 <nombre miembro> =datos;**

1. **Lectura de información de una estructura:** si se desea introducir la información en la estructura mediante el teclado, basta con emplear una sentencia de entrada utilizando el operador punto o puntero
2. **Recuperación de información de una estructura:** se recupera información de una estructura utilizando el operador de asignación o una sentencia “cout”. Igual que antes se puede emplear el operador punto o el operador puntero.

ESTRUCTURAS ANIDADAS

Una estructura puede contener otras estructuras llamadas “estructuras anidadas”. Las estructuras anidadas ahorran tiempo en la escritura de programas que utilizan estructuras similares. Se han de definir los miembros comunes solo una vez en su propia estructura y a continuación utilizar esa estructura como un miembro de otra estructura

* Se desea diseñar una estructura que contenga información de operaciones financieras. Esta estructura debe constar de un numero de cuenta, una cantidad de dinero, el tipo de operación (deposito o retirada de fondos) y la fecha y hora en que la operación se ha realizado. A fin de realizar el acceso correcto a los campos “día”, “mes” y “año”, así como “tiempo” (la hora y minutos) en que se efectuó la operación, se define una estructura “registro\_operacion” de la siguiente forma

**#include<iostream>**

**Enum tipo\_de\_operacion (DEPOSITO, RETIRADA);**

**Estruct registro\_operacion { long numero\_cuenta;**

**Float cantidad;**

**Tipo\_de\_operacion tipo\_operacion;**

**Struct {**

**int, mes, dia, año;**

**} fecha;**

**Struct {**

**Int horas, minutos;**

**} tiempo;**

**};**

**Void main ()**

**{**

**Struct registro\_operacion operación;**

**Operación.cantidad= 500.00;**

**Operación.tipo\_transaccion= DEPOSITO;**

**Operación.fecha.mes= 10;**

**Operación.fecha.dia= 25;**

**Operación.tiempo. horas= 8;**

**Operación.tiempo.minutos= 45;**

**}**

ARRAYS DE ESTRUCTURAS

Se puede crear un array de estructuras tal como se crea un array de otros tipos. Los arrays de estructuras son idóneos para almacenar un archivo completo de empleados, un archivo de inventario, o cualquier otro conjunto de datos que adapte a un formato de estructura. Mientras que los arrays proporcionan un medio practico de almacenar diversos valores del mismo tipo, los arrays de estructura le permiten almacenar juntos diversos valores de diferentes tipos, agrupados como estructuras

* **Arrays como miembros:** los miembros de las estructuras puede ser asimismo arrays. En este caso, será preciso extremar las precauciones cuando se accede a los elementos individuales del array; por ejemplo, una librería desea catalogar su inventario de libros. El siguiente programa crea un array de 100 estructuras, donde cada estructura contiene diversos tipos de variables, incluyendo arrays

**#include<iostream>**

**#include<stdio.h>**

**#include<ctype.h>**

**Struct inventario**

**{**

**Char titulo {25];**

**Char fecha\_pub [20];**

**Char autor [30];**

**Int nm;**

**Int pedido;**

**Float precio\_venta;**

**};**

**Main ()**

**{**

**Struct inventario libro [100];**

**Int total=0;**

**Char resp;**

**Do**

**{**

**Cout<<”total libros “<<(total+1)<<”\n”;**

**Cout<<”cual es el titulo?”;**

**Gets (libro[total].titulo);**

**Cout<<”cual es la echa de publicación?”;**

**Gets (libro[total].fecha\_pub);**

**Cout<<”quien es el autor?”;**

**Gets(libro[libro].autor);**

**Cout<<”cuantos libros existen?”;**

**Cin>>libro[total].num;**

**Cout<<”cuantos ejemplares existen pedidos?”;**

**Cin>>libro[total].pedido;**

**Cout<<”cual es el precio de ventas?”;**

**Cin>>libro[total].precio\_venta;**

**Fflush(stdin);**

**Cout<<”\nhay mas libro? (S/N)”;**

**Cin>>resp;**

**Fflush (stdin);**

**Resp=toupper(resp);**

**I(resp==’s’)**

**{**

**Total++;**

**Continue;**

**}**

**}while (resp==’s’);**

**Return 0;**

**}**

UTILIZACION DE ESTRUCTURAS COMO PARAMETROS

C++ permite pasar estructuras bien por valor o bien por referencia. si la estructura es grande, el tiempo necesario para copiar un parámetro “struct” a la pila puede ser prohibitivo. En tales casos, se debe considera el método de paso por referencia

**#include<iostream>**

**Struct inopersona{**

**Char nombre [20];**

**Char calle [30];**

**Char ciudad [25];**

**Char provincia [25];**

**Char codigopostal [5];**

**};**

**Void verinfo (infopersona datos)**

**{**

**Cout<<datos.nombre<<endl;**

**<<datos.calle<<endl;**

**<<datos.ciudad<<” “<<datos.provincia<<” “<<datos.codigopostal<<endl;**

**}**

**Void main (void)**

**{**

**Infopersona registrodatos = {“pepe luis mackoy”,**

**“3 de mayo”, “Cazorla”,**

**“jaen”, “63441”};**

**Verinfo(infopersona);**

**Cout<<”pulsa intro(enter)para continuar”;**

**Cin.get();**

**}**

**NOTA:** si desea pasar la estructura por referencia, necesita situar un operador de referencia & entre infopersona y datos en la cabecera de la función verinfo()

VISIBILIDAD DE UNA ESTRUCTURA: PUBLICA O PRIVADA

En C++ las estructuras se utilizan normalmente para contener tipos de datos diferentes, mientras que las clases se utilizan para contener datos y funciones. De hecho, C++ permite también estructuras que tienen miembros función y datos; diseñar una estructura “Punto” (coordenada en 3 dimensiones x, y, z) que proporcione dos funciones miembro “sumar()” y “restar()” que suman y restan respectivamente, dos objetos tipo Punto:

**#include<iostream>**

**Struct Punto**

**{**

**Doublé x, y, z;**

**Void sumar (const Punto& p1, const Punto& p2)**

**{**

**X=p1.x + p2.x;**

**Y=p1.y + p2.y;**

**Z=p1.z + p2.z;**

**}**

**Void restar (const Punto& p1, const Punto& p2)**

**{**

**X=p1.x - p2.x;**

**Y=p1.y - p2.y;**

**Z=p1.z - p2.z;**

**}**

**};**

**Void main ()**

**{**

**Punto p1, p2, p3;**

**P1.x=1.0; p1.y=2.0; p1.z=3.0;**

**P2.x=8.0; p2.y=9.0; p2.z=10.0;**

**P3.x=0.0; p2.y=0.0; p3.z=0.0;**

**Cout<<”antes: “<<endl;**

**Cout<<”p3.x: “<<p3.x<<endl;**

**Cout<<”p3.y: “<<p3.y<<endl;**

**Cout<<”p3.z: “<<p3.z<<endl;**

**P3.sumar(p1, p2);**

**Cout<<”despues: “<<endl;**

**Cout<<”p3.x: “<<p3.x<<endl;**

**Cout<<”p3.y: “<<p3.y<<endl;**

**Cout<<”p3.z: “<<p3.z<<endl;**

**}**

OPERADOR TYPEDEF

Un typedef permite a un programador crear un sinónimo de un tipo de dato definido por el usuario o integral ya existente

**Ypedef doublé longitud;**

**Longitud distancias (const punt&p, const punto& p2)**

**{**

**Longitud londitud=sqrt(r-cua);**

**Return longitude;**

**}**

PUNTEROS (APUNTADORES)

Una variable puntero es una variable que contiene direcciones de otras variables. Todas las variables vistas hasta este momento contienen valores de datos, por el contrario las variables punteros contienen valores que son direcciones de memoria donde se almacenan datos. En resumen, un puntero es una variable que contiene una dirección de memoria, y utilizando punteros su programa puede realizar muchas tareas que no seria posible utilizando tipo de datos estándar

DIRECCIONES Y REFERENCIAS

Cuando una variable se declara, se asocian tres atributos fundamentales con la misma: su nombre, su tipo y su dirección de memoria

El valor de una variable se accede por medio de su nombre. Por ejemplo, se puede imprimir el valor de “” con la sentencia “cout<<n;”.

A la dirección de la variable se accede por medio del operador de dirección “&”. Por ejemplo, se puede imprimir la dirección de “n” con la sentencia “cout<<&n;”

El operador de dirección “&” opera sobre el nombre de la variable para obtener sus direcciones. Tiene precedencia nivel 15 con el mismo nivel que el operador lógico “not” y el operador de pre-incremento “++”

**Main ()**

**{**

**Int n=75;**

**Cout<<”n= “ <<n <<endl;**

**Cout<<”&n = “<<&n<<endl;**

**}**