Acciones y funciones

1

Objetivos de aprendizaje
Dominando los temas del presente trabajo Usted podrá.

1. Entender la descomposición como forma de resolución de problemas.
2. Dar valor a la reusabilidad en búsqueda de la eficiencia en la escritura el código.
3. Establecer comunicación entre módulos.
4. Comprender las ventajas de la descomposición
5. Diferenciar acciones de funciones y los distintos tipos de parámetros y datos

Introducción

En este trabajo se analiza la descomposición como forma de alcanzar la solución de problemas. Una regla básica de la programación indica que si existe un programa de longitud L tal que L = L1 + L2, se dice que el esfuerzo de resolver L es mayor que la suma de los esfuerzos de resolución de L1 y L2, aun cuando haya que derivar esfuerzo para la integración de los módulos.

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

Antes de analizar las particularidades de las acciones y funciones es necesaria la definición de los términos que se utilizan.

Modularizacion

En general los problemas a resolver son complejos y extensos, puede incluso presentarse situaciones en que una parte del problema deba ser modificada para adaptarse a nuevos requerimientos. Se hace necesario conocer algunas herramientas que permitan facilitar la solución de estos problemas, la abstracción y la descomposición pueden ayudar a ello. La abstracción permitirá encontrar y representar lo relevante del problema y la descomposición se basa en el paradigma de ¨ dividir para vencer ¨. La descomposición tiene como objetivo dividir cada problema en subproblemas cada uno de los cuales será de más simple solución.
Es conveniente, e importante descomponer por varias razones:

- Favorece la comprensión. Una persona entiende un problema de características complejas partiendo la información, descomponiendolo. Por esto para comprender un problema complejo del mundo real es necesario dividirlo o modularizar.
- Favorece el trabajo en equipo. Cada programador recibe las especificaciones la tarea a realizar y las restricciones con las que debe manejarse.
- Favorece el mantenimiento. Las tareas involucradas en este mantenimiento, están vinculadas con detectar corregir errores, modificar y/o ampliar codigo. Esto es mucho mas

sencillo si se hace un análisis y control de una porción o modulo y luego se lo relaciona que atendiendo de una sola vez la totalidad del problema.

- Permite la reusabilidad del código. Siempre es deseable, de ser posible, hacer uso de código ya escrito.
- Permite además, como veremos, separar la lógica de la algoritmia con el tipo o estructura de dato involucrada. Alcanzando reusabilidad y diversidad de tipos tendiente a un criterio de programación mas genérica centrada en la algoritmia.

Módulos

Un problema debe ser descompuesto en subproblemas que se denominan módulos en los que cada uno tendrá una tarea especifica, bien definida y se comunicaran entre si adecuadamente para conseguir un objetivo común. Un modulo es un conjunto de instrucciones mas un conjunto de datos que realizan una tarea lógica.

Alcance de los datos

El desarrollo de un programa complejo con muchos módulos en los que en cada uno de ellos se deben definir los propios tipos de datos puede ocasionar algunos de los inconvenientes siguientes:

- Demasiados identificadores.
- Conflictos entre los nombres de los identificadores de cada modulo.
- Integridad de los datos, lo que implica que puedan usarse datos que tengan igual identificador pero que realicen tareas diferentes.

La solución a estos problemas se logra con una combinación entre ocultamiento de datos y uso de parámetros.

Datos locales y globales

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

Local y global puede ser sido utilizado de manera absoluta pero los módulos pueden anidarse las reglas que gobiernan el alcance de los identificadores son:

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

Ocultamiento y protección de datos

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

Parámetros

Son variables cuya característica principal es que se utilizan para transferir información entre módulos. En programas bien organizados toda información que viaja hacia o desde módulos se hace a través de parámetros.

Hay dos tipos de parámetros, los pasados por valor y los pasados por referencia o dirección.

Cuando existen datos compartidos entre módulos una solución es que un modulo pase una copia de esos datos al otro. En este caso el pasaje se denomina pasaje por valor. El modulo que recibe esta copia no puede efectuar ningún cambio sobre el dato original; el dato original se encuentra de este modo protegido de modificación.

Los parámetros pasados por referencia o dirección no envían una copia del dato sino envían la dirección de memoria donde el dato se encuentra por lo que tanto el proceso que lo llama como el proceso llamado pueden acceder a dicho dato para modificarlo.
Razones por la que es conveniente la utilización de parámetros sobre las variables globales.

Integridad de los datos

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

Protección de datos

Si una variable es local a un modulo se asegura que cualquier otro modulo fuera del alcance de la variable no la pueda ver y por lo tanto no la pueda modificar. Dado que las variables globales pueden ser accedidas desde distintos módulos, la solución para evitar modificaciones no deseadas es el pasaje como parámetros valor.

Uso de parámetros para retornar valores

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

Utilidad del uso de parámetros

El uso de parámetros independiza a cada modulo del nombre de los identificadores que utilizan los demás. En el caso de lenguajes fuertemente tipiados solo importa la correspondencia en cantidad tipo y orden entre los actuales del llamado y los formales de la implementación con independencia del nombre del identificador.

Reusabilidad

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

Acciones

El léxico establece el nivel de abstracción de un algoritmo. Es decir, introduce las variables, las constantes, los tipos de datos y las acciones con que se construye el algoritmo.
El concepto de acción está muy ligado al concepto de abstracción. Se analiza como abstracción por parametrización y abstracción por especificación.
Las acciones pueden desarrollar distintos cálculos y retornar cero, uno o mas resultados al programa que lo invoca. En general se los usa con una invocación a si mismo, no retornan valor en el nombre (solo podría retornar uno), en cambio retornan valores en los parámetros (los parámetros variables o enviados por referencia.

Utilización de acciones

Una *acción* es una secuencia de instrucciones que se identifica por un nombre y que puede ser invocada desde un algoritmo principal o desde otra acción. Cuando una acción es invocada desde algún punto de un algoritmo, el flujo de ejecución se traslada a la primera instrucción de la acción, entonces la acción se ejecuta hasta el final y cuando acaba, el flujo se traslada de nuevo a la instrucción del algoritmo que sigue a aquella que origino la invocación.
Una acción debe tener un efecto bien definido, lo que significa que debe ser cohesiva. El nombre de la acción es conveniente que evoque la tarea que realiza. Hay que definir acciones que sean aplicables a cualquier posible conjunto de valores de entrada y no a un valor concreto.

Entre una acción y el algoritmo que la invoca se debe producir una comunicación de valores: el algoritmo debe proporcionar los valores de entrada y la acción puede retornar el resultado, o puede modificar el estado de alguno de ellos.
Puede haber acciones en las que la comunicación se realice mediante variables globales definidas en el ámbito del algoritmo principal, que pueden ser manejadas por la acción. Pero esta no es la forma mas apropiada. Una acción, en principio, nunca debería acceder a variables globales.
Los *parámetros* son el mecanismo que posibilita escribir acciones generales, aplicables a cualquier valor de la entrada, e independientes del léxico del algoritmo.

Acciones con parámetros

Un *parámetro* es un tipo especial de variable que permite la comunicación entre una acción y el algoritmo que la invoca, ya sea que este pase a la acción un valor de entrada o bien que la acción devuelva el resultado al algoritmo, o que pasen ambas cosas simultáneamente.
A los parámetros que proporcionan un valor de entrada se los llaman *Parámetros dato*, y a los que, además de recoger un valor, retornan un resultado, se los llama *dato-resultado*. En la declaración de una acción, la lista de los parámetros se indica después del nombre de la acción entre paréntesis y separados por coma.

```
Nombre(dato tipo p1 ,dato-resultado tipo p2 )
```

Un parámetro también cuenta con una característica: la dirección de memoria en la que se realiza la transmisión de la información.
Se denomina *paso de parámetros* al modo en que se establece la comunicación entre los argumentos pasados a la acción desde el algoritmo y los parámetros de la acción; en la llamada se pasan los datos de entrada, y en el retorno se devuelven los resultados. Cada argumento se liga con el parámetro que ocupa la misma posición en la declaración de la acción y ambos deben coincidir en tipo.
En la invocación a una acción, el algoritmo debe proporcionar un valor para cada parámetro dato, mientras que debe indicar para cada parámetro dato-resultado (&) qué variable de su léxico recogerá el valor.

Abstracción y acciones

El término *abstracción* se refiere al proceso de eliminar detalles innecesarios en el dominio del problema y quedarse con aquello que es esencial para encontrar la solución. Como resultado de aplicar la abstracción, se obtiene un modelo que representa la realidad que nos ocupa. Este modelo no es la realidad, es una simplificación de esa realidad que establece un nivel de abstracción mas apropiado para razonar sobre el problema y encontrar la solución.
La abstracción también está relacionada con los lenguajes de programación. Estos ofrecen mecanismos de abstracción que determinan la forma de razonar de un programador.
El concepto de acción conjuga dos técnicas de abstracción que son la parametrización y la especificación. La parametrización es un mecanismo por el cual se generaliza una declaración para que no sea aplicado a un único caso, sino que sirva para cualquier valor que pueda tomar cierto parámetro.
La abstracción por especificación es la separación entre la especificación (el Qué) y la implementación (el cómo). En el caso de acciones, se refiere a distinguir entre la descripción de qué hace la acción y el cómo se la implementa. Una vez que se define una nueva acción, se las utiliza del mismo modo que si se tratase de una acción primitiva.
Del mismo modo que el algoritmo, las acciones se especifican mediante una precondición y una poscondición. La precondición establece las restricciones que satisfacen los parámetros (datos de entrada) para que se pueda ejecutar la acción, y la postcondición describe el resultado.

Cada acción debe ir siempre acompañada de una descripción textual de su efecto y de su precondición y pos condición. De esta forma, cualquier programador podría conocer qué hace y podría utilizarla sin conocer cómo lo hace.
El programador solo debe preocuparse por que se cumpla la precondición al invocar la acción y tendrá la certeza de que la acción cumplirá su objetivo.

Tipos de Parámetros

Una acción se comunica con el algoritmo que lo invoca a través de los parámetros. Es necesaria una comunicación en dos sentidos. El algoritmo debe proporcionar los datos de entrada que manipulará durante su ejecución y la acción debe retornar los resultados que obtiene.
El *tipo de parámetro* indica cómo los valores de los argumentos son ligados a los parámetros.
El tipo de parámetro *dato* solo permite que el parámetro pueda recibir el valor de un argumento mientras que el tipo de parámetro *dato-resultado* permite que el parámetro pueda recibir un valor y pueda retornar un resultado. En la declaración de una acción hay que indicar el tipo de cada parámetro. La declaración de una acción se la denomina *cabecera*.
Se denomina *paso por valor* al paso de parámetros que corresponda al tipo de parámetro dato y *paso por referencia* al que corresponda al tipo de parámetro dato-resultado.
Parámetro dato (o parámetro de entrada)
El valor del argumento es asignado al parámetro en el momento de la llamada. El argumento puede ser una expresión del mismo tipo de dato que el parámetro. Se trata de una comunicación unidireccional: solamente se transmite información desde el punto del llamado hacia la acción.
Una regla de buen estilo de programación es no modificar el valor de parámetros tipo dato, aunque ello no tenga efecto fuera de la acción.
Parámetro dato-resultado (o parámetro de entrada y salida)
El valor del argumento es asignado al parámetro en el momento de la llamada y al final de la ejecución el valor del parámetro es asignado al argumento. Se trata de una comunicación bidireccional. Si la ejecución de la acción provoca un cambio en el valor del parámetro, en el momento del retorno el argumento tendrá el valor del parámetro al finalizar la ejecución.
Un argumento para un parámetro dato-resultado debe ser una variable del mismo tipo de dato que el parámetro y no puede ser una expresión.
Acción para el intercambio de dos variables: La acción recibe dos identificadores con dos valores y debe retornar los identificadores con los valores cambiados

```
Intercambiar(dato-resultado a, b : Entero) : una acción
PRE { a, b : Entero, a = A, b = B }
POST { a = B, b = A }
LÉXICO
      t : Entero
ALGORITMO
t = a;
a = b;
b = t
FIN
```

Estos son los parámetros, que como deber ser modificados sus valores por la acción se definen como Dato-Resultado, son las variables que intercambian información entre el programa principal y el modulo. Son

Este identificador se necesita como variable auxiliar para poder hacer el intercambio. Su valor no lo necesita el programa que invoca a la acción, solo interesa su visibilidad en el modulo por tanto no es

Ejemplo en C++.

/* Funcion que no retorna valor en su nombre, el tipo de retorno void es lo que lo indica. La función tiene dos parámetros que son pasados por referencia o dirección, el & delante del

parámetro es el que lo indica. Al ser enviados por referencia el programa que lo invoca recibe los identificadores con los valores intercambiados */

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

Beneficios del uso de acciones

En la actualidad, las clases de los lenguajes orientados a objetos son los mecanismos más adecuados para estructurar los programas. No obstante, las acciones no desaparecen con estos nuevos mecanismos porque dentro de cada clase se encuentran acciones.

Una acción tiene cuatro propiedades esenciales. Ellas son:

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

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

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

La función desarrollada tiene la propiedad de la reusabilidad ya mencionada, es decir, puede ser invocada en distintos puntos del programa con pares de identificadores de tipo entero y producirá el intercambio. Sin embargo para que pueda producirse el intercambio utilizando esta acción el requerimiento es que los parámetros sean de tipo int.

Algoritmos de UTNFRBA utilizara para las implementaciones C++. Este lenguaje agrega una característica de reutilización de código realmente poderosa que es el concepto de plantilla. Las plantillas de funciones, que son las que utilizaremos son de gran utilidad para evitar escribir código cuando las acciones deben realizar operaciones idénticas pero con distintos tipos de datos.

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

typename o class. Nosotros usaremos **typename**. Por ejemplo:

template <typename Tn > si solo involucra un tipo de dato

o

template < typename R, typename T> si involucra dos tipos de dato diferentes

La función del ejemplo tiene un solo tipo de dato que es int, si quisiéramos hacer el intercambio utilizando variables de otro tipo, por ejemplo doublé, también tendríamos un único tipo en toda la acción. En lugar de colocar un ti´po definido optaremos por un tipo genético al que llamaremos T.

La función quedaría:

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

El tipo genérico es T por lo cual en cada aparición de int de la función original fue reemplado por el tipo genérico y ahora la misma acción es valida no solo para diferentes pares de valores (reusabilidad) sino además para distintos tipos de datos (polimorfismo)

Funciones

Si el propósito es calcular un valor a partir de otros que se pasan con argumentos y se utilizan acciones, habrá que definir un parámetro dato-resultado para que retorne el valor calculado. Las acciones que retornan valor en los parámetros no pueden ser utilizadas en una expresión.

Para resolver este problema, los lenguajes de programación incorporan el concepto de función. Las funciones devuelven un único valor. La función supone extender el conjunto de operadores primitivos. Se invocan en una expresión.

En cada declaración se especifica el nombre de la función, la lista de los parámetros y finalmente el tipo de valor que retorna la función.

$Nombre\_funcion\ (par_1 : td_1\ ;\ \ldots\ ;\ par_n : td_n) : tr : una\ función$

Una función no debe tener efectos laterales. Es decir, debe limitarse a calcular un valor y no modificar ninguno de los que se describen en el momento de la invocación.

Las acciones se utilizan para extender el conjunto de acciones primitivas. Las funciones permiten extender el conjunto de funciones u operadores primitivos. Siempre deben utilizarse dentro de una expresión.

Función que obtenga el mayor de tres números

```
Max(a, b, c : Entero) → Entero : una función
        ALGORITMO
          SI (a >= b) y (a >= c)
  ENTONCES
Max = a
  SINO SI (b >= a) y (b >= c)
  ENTONCES
        Max = b
  SINO
Max = c
        FIN_SI
        FIN_SI
       FIN.
```

Las funciones siempre retornan un unico valor. El identificador del nombre de la funcion se puede utilizar en una expresión como cualquier identificador del tipo que retorna. Las funciones pueden retornar un escalar un apuntadoro un registro

```
/definicion de la plantilla de la función Max
template < typename T>
T Max ( T a, T b, T b) \*          vea que la función retorna un valor de tipo T y los parámetros no
                                   están pasados por referencia*\
{// comienzo del bloque de declaración de la funcion
        T t = a;  //declaración de una variable auxiliar t para contener el maximo
        if (b > t) t = b;
        if (c > t) t = c;
        return t;          //retorna en t el valor del maximo.
} // fin del bloque de la funcion
```

Concepto de recursividad:

Es un proceso que se basa en su propia definición. Una función puede invocarse a sí misma como parte de los tratamientos de cálculo que necesita para hacer su tarea

Parte de instancias complejas y las define en términos de instancias más simples del mismo problema, llegando a un punto donde las instancias más simples son definidas explícitamente.

Define el problema en términos de un problema más simple de la misma naturaleza.

Debe disminuir el espacio del problema en cada llamada recursiva

Hay una instancia particular que se conoce como caso base o caso degenerado

Divide el problema original en subproblemas más pequeños. Cuando es lo suficientemente chico se resuelve directamente y se combinan soluciones del subproblema hasta que queda resuelto el problema

Tiene:

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

Funcion Factorial

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

Instancias que permanecen en memoria:

Funcion PotenciaNatural

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

Funcion DivisionNatural

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

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

Eliminación de la recursividad

Toda función recursiva puede ser transformada a la forma iterativa mediante una pila (LIFO).

La recursión es una abstracción, una construcción artificial que brindan los compiladores como herramienta extra para resolver problemas, sobretodo aquellos que son de naturaleza recursiva.

Resumen:

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

Desde el lenguaje

2

Objetivos de aprendizaje
Implementacion en C++.

1. Implementacion de funciones.
2. Conceptos de reusabilidad.
3. Punteros a funciones

RESUMIENDO

- La experiencia demuestra que la mejor forma de desarrollar y mantener un programa extenso es construirlo a partir de componentes simples y pequeños. Estos se implementan con funciones
- En general los programas se escriben mediante la combinación de funciones provistas y desarrolladas.
- Las funciones permiten modular para separar las tareas en unidades mas pequeñas
- Las instrucciones en los cuerpos de las funciones se escriben solo una vez, se pueden reutilizar desde varias ubicaciones en el programa y están ocultas a las demás funciones.
- Los prototipos de las funciones de biblioteca o funciones globales se colocan en archivos de encabezado y se pueden reutilizar en cualquier programa que los incluya, creando un enlace con el código objeto de la función.
- El compilador hace referencia al prototipo de la función para comprobar que las llamadas a la función tengan el número y tipo de argumentos correcto, estén en el orden adecuado y que el valor de retorno se pueda utilizar de manera correcta en la expresión que llamo a la función.
- Hay tres formas de devolver el control al punto en que se invoco a la función. Si la función no retorna valor se puede volver al llegas a la llave derecha de fin de función, o mediante la ejecución de la instrucción return;. Si la función retorna un valor se vuelve con la instrucción return expresión; donde la expresión debe ser del tipo del valor de la devolución.
- Un prototipo de función indica al compiladorel nombre de la función, el tipo de dato devuelto por la función, el número de parámetros que la función espera recibir, los tipos de esos parámetros y el orden en que se los espera.
- La porción de un prototipo que incluye el nombre de la función y los tipos de sus argumentos se llama firma de la función.

- Una característica importante de los prototipos es la cohersion de argumentos; es decir, obligar que los argumentos tengan los tipos especificados por las declaraciones de los parámetros.
- La biblioteca estándar esta dividida en varias porciones, cada una con su propio archivo de encabezado.
- Los archivos de encabezado instruyen al compilador acerca de cómo interconectarse con los componentes de la biblioteca y los componentes escritos por el usuario.
- En C++ una lista de parámetros vacía se especifica con void o con nada entre paréntesis.
- Dos formas de pasar argumentos: por valor o, por referencia
- Cuando se pasa por valor se crea una copia del valor del argumento. Las modificaciones de la copia no afectan el valor de la variable que hizo la llamada.
- En el pasaje por referencia la función que hace la llamada proporciona a la función que llamo la habilidad de acceder directamente a los datos de la primera y de modificarlos en caso que la función que se llamo asi lo decida.
- El parámetro por referencia es un alias para su correspondiente argumento en la llamada a una función.
- Para indicar que un parámetro se pasa por referencia simplemente se debe agregar & después del tipo.
- Una vez que se declara una referencia como alias para otra variable, todas las operaciones que supuestamente se realizan en el alias, en realidad se realizan en la variable original.
- C++ ofrece un operador de resolución de ámbito binario :: para acceder a una variable global cuando una variable local con el mismo nombre se encuentra en el alcance.
- C++ permite definir funciones con el mismo nombre siempre y cuando tengan distinta firma. Esto se conoce como sobrecarga de funciones.
- La sobrecarga, en general, se utilizan para realizar operaciones similares que impliquen distintas logicas o distintos tipos de datos.
- Si la lógica y las operaciones son idénticas para cada tipo de dato, la sobrecarga se puede hacer en forma mas compacta utilizando plantillas de funciones.
- El programador escribe una sola definición de la plantilla de función. Dados los tipos de los argumentos de las llamadas C++ genera, de manera automática especializaciones de la plantilla de función separadas para manejar cada tipo de llamada de manera apropiada. Por ende, al definir una plantilla de función en esencia se define una familia de funciones sobrecargadas
- Un puntero a una función contiene la dirección de memoria a la función. Asi como el nombre de un arreglo es la dirección de memoria del primer elemento del mismo. En forma similar el nombre de una función es la dirección en memoria del código que realiza la función.
- Los apuntadores a funciones se pueden pasar a las funciones, devolver de las funciones, asignar a otros apuntadores a funciones y usarse para llamar a la función subyacente.
- Por ejemplo bool (*compara)(int, int) especifica un apuntador a la función. La palabra reservada bool indica que la función a la que apunta retorna un valor bool. El texto (*compara) indica el nombre del apuntador a la función. (* indica que compara es un apuntados. El texto (int, int) indica que la función a la que apunta compara recibe dos argumentos de tipo int. Los paréntesis se necesitan alrededor de *compara, debe ser (*compara) para indicar que es un apuntador. De no incluirse y si la declaración es bool*compara(int,int) se esta declarando una función cuyo nombre es compara y retorna un puntero a bool y no un puntero a una función

Propósitos de las funciones

Permite Descomposición como forma de alcanzar la solución.

SI L = L 1 + L 2

Entonces

Esfuerzo$_{(L)}$ > Esfuerzo$_{(L1)}$ + Esfuerzo $_{(L2)}$

Promueve la Modularidad como forma de favorecer

La comprension - El trabajo en equipo.

Mantenimiento – Reusabilidad.

Facilitar código – Evita repeticiones.

Permite integridad y proteccion asi como separar la logica de la algoritmia y del tipo de dato (plantillas).

Programación modular-estructurada

- La programación estructurada propone dividir el problema en módulos
- Cada módulo permite aislar mejor el problema
- Todo programa tiene una función main ()
- La función main invoca a las otras funciones
- Cada una con una tarea determinada.
- return retorna al punto de invocación

Estructura de una función

Declaración

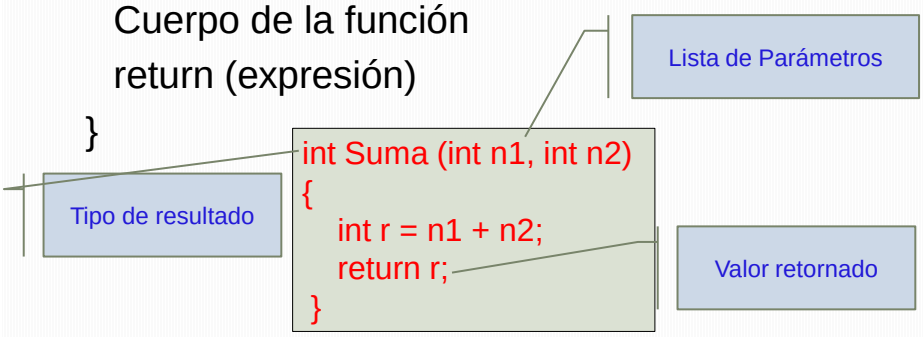
Tipo_Retorno Nombre(Lista_Parametros)

{

Cuerpo de la función

return (expresión)

}



Declaraciones

Nombre

Reglas de los identificadores

Tipo de retorno

Escalar, puntero, struct , void

Parámetros

Sin parámetros, void, con parámetros

F1(); F2(void); F3(int a, int& b)

Resultados

return; return (expresión); return expresión

Prototipos de las funciones

Tipo_Retorno Nombre (DeclaracionParametros)

int F1 (void); Sin parámetros

int F2 (int); con un parámetro valor de tipo int

int F3(int, &int); dos parametros int uno, valor y otro variable

Parámetros de las funciones

Pasaje por valor	Pasaje por referencia

Declaración
int Suma(*const* int a, int b)
{
  return (a+b);
}
 	
Reciben copias de los argumentos y los protegen de modificaciones
const añadiria seguridad en caso de ser de solo lectura. No permite modificarlo
Invocación
……
Int c = Suma(x,y);
……
	Declaración
	void Intercambio(int& a, int& b)
	{
	  int c = a;
	  a = b;
	  b = c;
	  return;
	};
	Reciben direcciones de memorias de los argumentos y los modifican
	
	Invocación
	…..
	Intercambio (x,y);
	…..

Arreglos y cadenas c/ parámetros

Siempre se pasan por referencia (sin &)

#define MAX 100

int Vec[MAX], Mat[MAX][MAX];

char S[MAX + 1]

Declaración	Invocación
void F1(int V[MAX]) void F1(int V[]) void F2(int M[MAX][MAX]) void F2(int M[][MAX]) void F3(char s[]) void F3(char * s)	F1(Vec) F2(Mat) F3(S)

Plantillas de funciones y tipos

Para implementar algoritmos independientes del tipo de dato sobre el que actúan. Permiten genericidad: definir sin especificar el tipo de dato de uno o mas miembros: parámetros, valor de retorno.

Sintaxis

template <typename T>

Los tipo de parametros y argumentos aparecen entre < y >

Ejemplo de plantillas

Sin Plantilla	Con Plantilla

El Nodo
struct Nodo {
int info;
Nodo * sig;
};
La Declaración
 	
void push(Nodo*& p, int v)
{
Nodo* nuevo = **new** Nodo();
nuevo->info = v;
nuevo->sig = p;
p = nuevo;
}
La Invocación
	El Nodo
	template <**typename** T> **struct** Nodo {
	T info;
	Nodo<T>* sig;
	};
	La Declaración
	template <**typename** T> **void**
	push(Nodo<T>*& p, T v)
	{
	Nodo<T>* nuevo = **new** Nodo<T>();
	nuevo->info = v;
	nuevo->sig = p;
	p = nuevo;
	}
	La Invocación

Nodo * L = NULL; push(L, 1);	Nodo<int> * L = NULL; push<int>(L,1);

Punteros a funciones

Se pueden crear apuntadores a funciones, en lugar de direccionar datos los punteros a funciones apuntan a código ejecutable.

Un puntero a una función es un apuntador cuyo valor es el nombre de la función.

Sintaxis

TipoRetorno (*PunteroFuncion)(ListaParametros)

```
int f(int);          //declara la funcion f
int (*pf)(int);      // define pf a funcion int con argumento int
pf = f;              // asigna la direccon de f a pf
```

Los apuntadores permiten pasar una función como argumento a otra función

Sin Plantilla	Con plantilla y función como parámetro

La declaración
Nodo* Insertar (Nodo * L, int x)
 	
{
  Nodo * q = new Nodo();
  q -> info = x;
  if (L==NULL|| x<L->info {
  q->sig = L; return q; }
  else
  Nodo *p =L
  while(p->sig!=NULL &&
  x>p->sig->info)
  p = p->sig;
  q->sig = p->sig;
  p->sig = q; return q;
}
 	
La invocación
Nodo * L = NULL;
Insertar (L, 1);
	La declaración
	template <typename T> Nodo<T>*
	Insertar(Nodo<T>*& L, T x, int (*criterio)(T,T))
	
	 Nodo**<T>** * q = new Nodo**<T>** ();
	  q -> info = x;
	  if (L==NULL|| criterio(x, L->info)<0 {
	  q->sig = L; return q; }
	  else
	  Nodo**<T>** *p =L
	  while(p->sig!=NULL && criterio(x, p->info)>0)
	  x>p->sig->info)
	  p = p->sig;
	  q->sig = p-sig;
	  p->sig = q; return q;
	int CritAsc(Alumno a1, Alumno a2)
	{ return a1.leg – a2.leg;}
	La Invocación
	Nodo<Alumno> * L = NULL;
	Alumno a; ……
	Insertar<Alumno>(L,a,CritrAsc);….

Insertar en Lista Circular

```
template <typename T> void Circular(Nodo<T>*& Lc, T x)
{
        Nodo<T> * q = new Nodo<T> ();
        q -> info = x; q->sig = q //apunta a si mismo
        if (Lc != NULL)
        {       //Si no es vacía inserta en el nodo anterior a la lista
                q->sig = Lc->sig; //enlaza el nuevo con el primer nodo
                Lc->sig = q; // pone el nuevo en ultimo lugar
         }
        Lc = q;//la lista tiene la dirección del ultimo nodo insertado
    return;
 }
```

El puntero a la lista apunta al ultimo nodo insertado.

Instancias de la función: Invocación

Función que retorna valor → en expresion

Identif = NombreFuncion (lista de argumentos)

Los argumentos serán ValorL si se relacionan con parámetros variables o expresiones con parametros valor

Funcion que retorna void → invocacion a si misma

NombreFuncion(lista de argumentos)

Los argumentos con identico criterio a lo anterior

Instancias de la función: Prototipo

Función que retorna valor

Tipo retorno = NombreFuncion (lista de tipo de parametro)

int suma (int, int);

int Sup(int, int, &int);

Valor de retorno escalar, puntero o struct

Funcion que retorna void

Solo cambia el tipo de retorno a void

Instancias de la función: declaración

Función que retorna valor

Tipo de retorno = NombreFuncion (lista de parámetros)

Los parametros pueden ser pasados por valor se indica Tipo de dato Identificador o por dirección, se indica Tipo de dato& Identificador

Funcion que no retorna valor solo modifica valor de retorno a void