**Programación modular**

**Introdución**

* A **programación modular** é a metodoloxía de deseño consistente en dividir programas en módulos, funcións lóxicas perfectamente diferenciables.
* Require a existencia dun módulo principal que controle ós demais.
* **Módulos de definición e implementación:** Partes que constitúen os módulos de biblioteca.
* **Módulos internos:** Van dentro do mesmo arquivo ca o programa principal ou nos módulos de implementación. Son os procedementos que veremos de seguido.

**Funcións e procedementos**

* **Función:** Subprograma que realiza unha operación significativa sobre un conxunto de parámetros. Devolven un valor mediante return.
  + Sempre ten argumentos. Os seus argumentos serán valores dunha variable cos que se invoca a función.
  + Pode conter varios returns para varias condicións distintas. Ao chegar a un finaliza a función.
  + Defínese como expresión abstracta.
* **Procedemento:** Non devolve un valor.
  + Pode ter ou non argumentos.
  + Cando emprega unha chamada por referencia, este argumento debe ser un punteiro. Desta forma, pásase a posición en memoria da variable.
  + Finaliza cando acaba o código ou cando se chega a un return (mellor non usalo, non devolve parámetros).
  + Defínese como acción abstracta.

**Funcións e procedementos estándar**

* Aquelas que forman parte do compilador (só se se inclúe o módulo relevante mediante #include). Por exemplo, as funcións de **<math.h>**,.
  + Devolven un *domain error* se a entrada está fóra do dominio, e un ±HUGE\_VAL se é demasiado grande.

**Variables**

* As variables declaradas dentro dunha función ou procedemento son variables locais. Non existen fóra da función, e existen só durante a execución da mesma.
  + Os parámetros introducidos a unha función son tamén locais.
* As variables declaradas na sección declarativa do programa principal son variables globais, e pódese acceder a elas dende calqueira módulo do arquivo.
* Dúas variables locais a distintas funcións poden ter o mesmo nome. Porén, se unha variable local ten o mesmo nome que unha global, o nome pasa a referirse só á local.

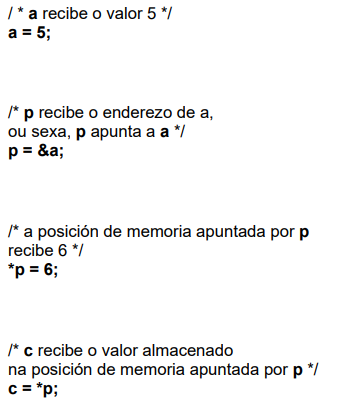
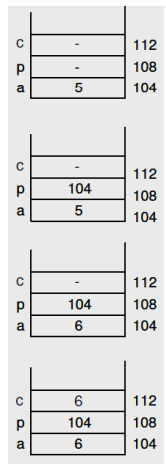
**Uso de memoria**

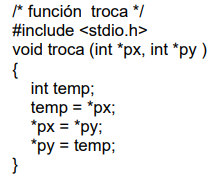
* Durante a compilación non é posible determinar todos os requerimentos de memoria. Distinguimos entre 3 partes da memoria do ordenador:
  + **Memoria estática:** Empregado para variables globais, estáticas[[1]](#footnote-0) e instrucións. Asignada durante compilación.
  + **Memoria dinámica:** Empregado para variables locais, e outros datos relacionados coa chamada de funcións.
    - O seu tamaño aumenta ou diminúe durante a execución, tomando parte dunha memoria libre asignada, se é necesario.
    - Almacena o punto ao que volver tras executar un subprograma, o estado da execución nese punto, os argumentos e variables locais do subprograma e os seus valores de return.
  + **Memoria dinámica montón:** Variables descoñecidas (tanto o seu tamaño coma a súa existencia) durante compilación.
    - É unha memoria dinámica pero non estruturada. A reserva é especificada polo programador e a liberación, dependendo da linguaxe, pode ser automática ou manual.
    - Ao montón accédese mediante punteiros ou referencias.

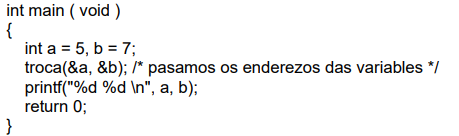
**Paso de argumentos**

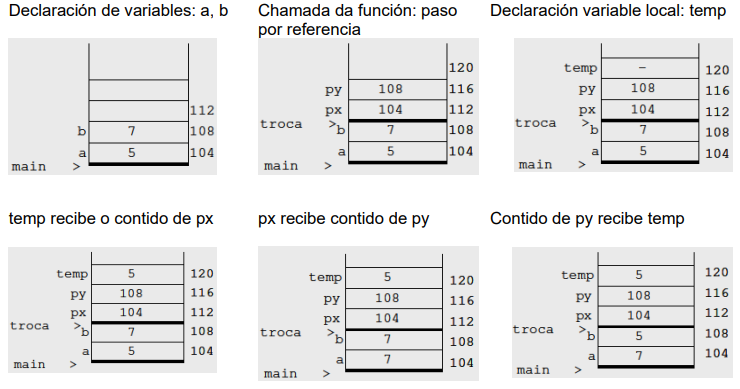
* Comunican os módulos entre si.
* **Parámetros formais:** Sitúanse na definición do módulo.
* **Parámetros actuais:** Sitúanse na invocación do módulo.
* Diferenciamos entre parámetros de entrada, saída e entrada/saída (modifícanse na función).
* Todos se poden empregar e pasar por valor ou por referencia.
  + Por valor: Pásanse aos parámetros formais como copias dos valores dos argumentos. Se se modifican estas copias as orixinaisnon cambian.
  + Por referencia: Pásanse ao módulo os enderezos de memoria das variables orixinais. Ao modificarse altéranse as variables orixinais. Pódense manipular mediante punteiros.

**Punteiros**

* Un punteiro é unha variable que fai referencia a unha **posición de memoria.**
* int \*p; define unha variable punteiro a enteiro.
* Para traballar con punteiros empréganse:
  + Operador unario & (“enderezo de”). Aplícase sobre unha variable para devolver a súa posición de memoria.
  + Operador unario \* (“contido de”). Aplícase sobre un punteiro para acceder ao seu contido.

* As chamadas por referencia empréganse para obter valores dos cálculos que faga un procedemento, e para aforrar memoria e tempo.
* Cando un módulo g chama a outro módulo f, f non pode alterar valores de variables de g.
  + Porén, g pode pasarlle a f os valores dos enderezos de memoria onde se almacenan as variables de g. Desta forma, f pode alterar indirectamente os valores destas variables. 





**Recomendacións sobre parámetros**

* Evitar parámetros de saía con funcións, empregando no seu lugar a sentencia return
* Se o módulo precisa devolver máis dun valor, definilo como procedemento (void) e devolver eses valores mediante parámetros de saída ou entrada/saída
* Non facer uso directo de variables globais, pasalas como parámetros de entrada

**Vectores (listas) como parámetros**

* Os vectores poden ser utilizados como parámetros en funcións e procedementos, pero unha función non pode devolver un vector.
  + Cando unha función/procedemento ten un parámetro que é un vector, pásase por referencia. No canto de devolver o valor, a función modifica o valor dun vector ‘resultado’ que se declara como un parámetro.
  + Cando se crea un vector para empregar en funcións, non se especifica a súa lonxitude na declaración da función. Escribiríase por exemplo **void** suma (float a[],b[],c[])
* As **matrices** tamén se poden pasar por referencia, pero na declaración do parámetro formal aparece un argumento estático: **void** suma (float mat[]**[4]**)
  + É preciso coñecer o segundo parámetro para saber cantos elementos hai en cada fila. Desta forma o sistema pode acceder correctamente ao elemento requirido.
  + Por exemplo, o elemento mat[i][j] é o elemento k=i\*n+j, onde n é o número de columnas. Entón, é preciso declarar este dato.

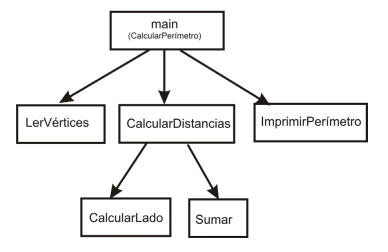
**Rexistros como parámetros**

* Os rexistros tamén poden ser utilizados como parámetros. Poden ser pasados tanto por valor como por referencia.
* Débese declarar o nome da estructura na declaración da función. Exemplo: **void** imprime (struct Punto p), onde ‘Punto’ é un rexistro que definimos previamente.
* Dentro da función accedemos ás variables do rexisto mediante un punto: p.x

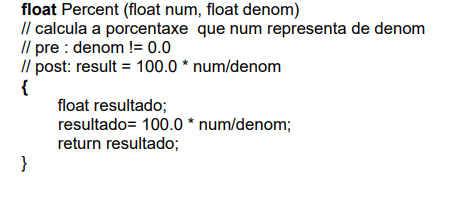
**Activación dun subprograma**

* Durante a activación dun subprograma execútanse os seguintes pasos:
  + Dende un módulo P chámase ao módulo Q.
    - Detense P, gárdase a súa información de estado
    - Asígnaselle memoria aos parámetros formais e variables locais de Q.
    - Se se fai unha chamada por referencia, cópiase o enderezo en memoria.
  + Empeza a executarse Q
  + Remata a execución de Q. Se existen, os valores de return pasan á pila.
  + Restaurase o estado previo á chamada de Q e continúa P.
  + Elimínanse os datos innecesarios da pila.

**Deseño modular**

* Un módulo é un fragmento dun programa que se desenvolve de forma relativamente independente.
* De estar ben desinado, débese poder empregar nun programa distinto só con entender o que fai.
  + Un programa descomposto en módulos escríbese como un conxunto de arquivos fonte que se poden compilar por separado correctamente.
* A técnica de programación estructurada descendente dos DFD pódese ampliar para contemplar a descomposición modular. A partir dun DDF podemos obter un **Diagrama de Descomposición Funcional (DDF)**
  + Xera unha descomposición de 2º nivel: nuns casos identificamos módulos cun proceso, noutros varios procesos agrúpanse nun módulo e noutros un proceso crea varios módulos.
  + O DDF consiste en establecer unha estrutura xerárquica entre os módulos:
    - Cada módulo é un rectángulo.
    - Cando un módulo invoca a outro, aparece máis alto na xerarquía, e sinálao cunha frecha.
    - Todos os módulos deben ter pais, salvo o principal. Non teñen que ter fillos necesariamente.
    - 
* Nesta descomposición existe un módulo que é o principal (correspóndese co nivel 0 no DFD). A execución do programa principal equivale á do programa completo.
* Correspóndelle ao programador considerar qué procesos merece a pena expresar como módulos. Un programa con moitos módulos innecesarios é máis lento, pero se unha parte do código se repite varias veces é eficiente expresala como módulo.

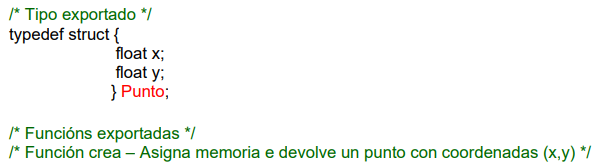
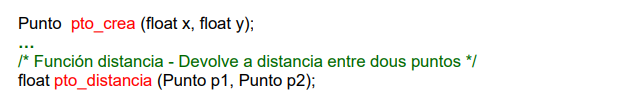
**Programación por contrato**

* É preciso que todas as funcións e procedementos esten correctamente documentados, incluíndo información sobre o seu propósito, precondicións e poscondicións.
* Exemplo: un programa que calcula un porcentaxe de dous numeros. É preciso especificar que o denominador non pode ser 0, e tamén o resultado esperado do programa:
* 
* Tamén sería posible incluír un código que comprobase se as variables son válidas na propia función.
* Desta forma, as precondicións e poscondicións son o ‘contrato’ asinado entre programadores. De ser incumprido, poderán aparecer errores.

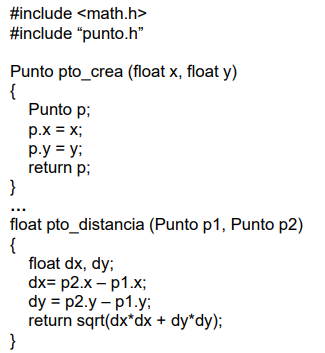
**Bibliotecas**

* Os módulos que procuremos empregar en distintos programas deben ser desinados e compilados independentemente de calqueira aplicación concreta.
* Denomínase ‘ocultación’ a característica de independencia entre un módulo e o seu programa. Require baixo acoplamento entre módulos e alta cohesión dentro de cada módulo.
  + O ‘acoplamento’ entre módulos indica cantos elementos dun módulo é preciso ter en conta para empregar ese módulo noutro. É preciso minimizalo.
  + A cohesión indica o grado de relación entre os distintos elementos dun módulo, é preciso maximizalo.
* Un módulo débese poder usar dende fóra sabendo o que fai sen necesariamente saber como o fai.
* Un módulo de biblioteca consta de dúas partes: especificación (ou **interface**) e a realización (ou **implementación**).

**Interface dun módulo**

* A interface dun módulo é un arquivo que só contén os prototipos de funcións/procedementos que ofrece o módulo e o tipo de datos que exporta.
* En C, un arquivo de interface ten o mesmo nome ca o módulo e leva a extensión **.h**. Denomínanse ‘arquivos cabeceira’.
* Inclúense mediante #include <arquivo.h> para bibliotecas estándares ou #include <arquivo.h> para arquivos creados polo usurario.
* Exemplo de arquivo de interface:
  + 
  + 

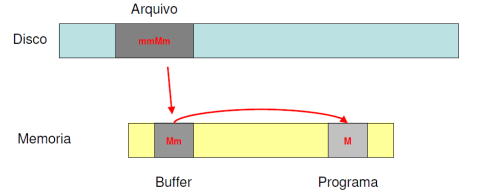
**Implementación dun módulo**

* A implementación dun módulo é o arquivo que contén a realización das funcións e procedementos declaradas na interface. Neste documento escríbense o código das funcións e procedementos.
* Leva o mesmo nome ca o módulo, coa extensión **.c**
* ****

**Uso dun módulo**

* Os elementos dun módulo pódense empregar dentro de outro mediante a sentencia #**include** <arquivo.h> se é unha biblioteca estándar, #**include** “arquivo.h” se é do usuario.
* Os módulos compílanse de forma separada. Desta forma, cada módulo ao compilarse da lugar a un arquivo obxecto .o ou .obj.
  + Isto permite que, se se cambian só algúns módulos, sño é preciso recompilar eses arquivos, aforrando tempo.

**Arquivos**

* Os ficheiros permiten almacenar información de forma permanente, habitualmente en soporte magnético ou óptico.
* Os arquivos son estruturas de datos en memoria externa que poden ser manipulados mediante a relación establecida entre esta memoria externa e as variables ficheiro definidas en memoria interna.
* O sistema operativo ofrece os seguintes servizos:
  + Apertura do arquivo, establecendo unha conexión cun punteiro de tipo ficheiro e preparando o buffer de memoria
  + Lectura e escritura do arquivo, modificando e lendo os datos do buffer.
  + Peche do arquivo, transfirindo os datos do buffer ao disco.
  + 

**Empregar arquivos**

* Para abrir un arquivo emprégase a función **FILE\* fopen (char\* nome\_arquivo, char\* modo);**
  + Devolve un punteiro ao tipo FILE.
  + nome\_arquivo pode ser relativo ao directorio do programa ou incluír directorios dende a raíz (C/Users…)
  + ‘modo’ pode ser:
    - w para escritura. Se non existe créase un novo, se existe formatéase
    - a para engadir ao final do arquivo
    - t para texto e b para binario , pódense combinar cos demais
    - r+ e w+ para lectura+escritura nun arquivo existente ou novo, respectivamente.
* Para pechar un arquivo emprégase a función **int fclose (FILE\* fp)**;
  + fp é o punteiro do arquivo a pechar.
  + Devolve a constante EOF en caso de erro.

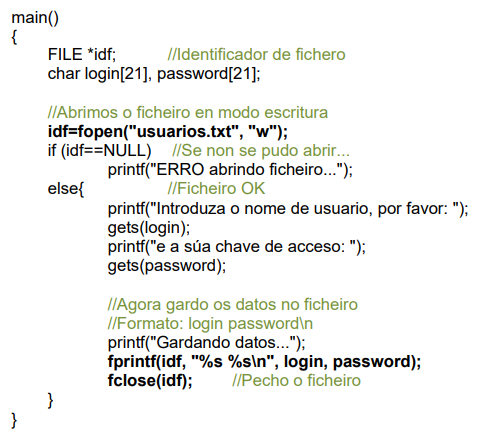
**Arquivos de acceso secuencial**

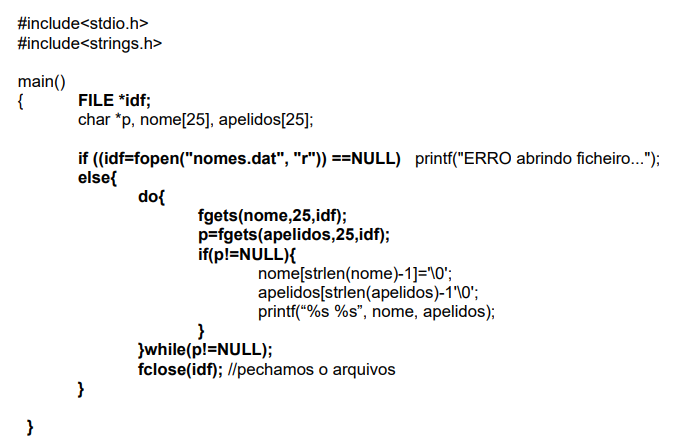
* Nun **ficheiro de acceso secuencial** o acceso a un dato esixe o acceso previo a todos os datos interiores.
  + Modos diferentes para escritura e lectura
    - No modo escritura o ficheiro gárdase dende 0, sobreescribindo os datos previos se non se volven a introducir.
    - Nalgúns casos permítese engadir datos sen sobreescribir
  + Formados por carácteres, organizados de forma lóxica
  + As unidades estándar ábrense e péchanse de forma automática ao comezo e final do programa.

**Funcións de ler arquivos en modo texto**

* **int fscanf(FILE\* fp, char\* formato, …);**
  + Transfire datos para a memoria e avanza o cursor cara o próximo dato.
  + Parámetros: punteiro ao arquivo, formato e lista de enderezos a variables que almacenarán os valores lidos.
  + Devolve o número de datos lidos con éxito. -1 en caso de erro.
* **int fgetc (FILE\* fp);**
  + Recolle un carácter do arquivo e avanza o cursor cara o próximo carácter.
  + O único parámetro e o punteiro ao arquivo
  + Devolve o código do carácter lido ou a constante EOF se chegou ao fin do arquivo
* **char\* fgets(char\*s, int n, FILE\*fp)**
  + Le unha cadea de carácteres ata atopar un \n **ou** cando chega ao máximo número de caracteres indicado por n.
  + Parámetros: cadea de caracteres que almacenará o lido, número máximo de caracteres, punteiro ao arquivo
  + Devolve un punteiro á cadea de carácteres ou NULL en caso de erro.

**Funcións de escribir en arquivos en modo texto**

* **int fprint(FILE\* fp, char\* formato, …)**
  + Similar á función printf, pero escr,,íbeo nun arqivo.
  + Parámetros: punteiro ao arquivo no que grabar os datos, formato e lista de variables que conteñen os datos a gardar.
  + Devolve o número de bytes escritos.
* **int fputc(int c, FILE\* fp)**
  + Escribe un carácter no arquivo.
  + Parámetros: código do carácter a escribir, punteiro ao arquivo
  + Devolve o carácter escrito ou EOF en caso de erro.

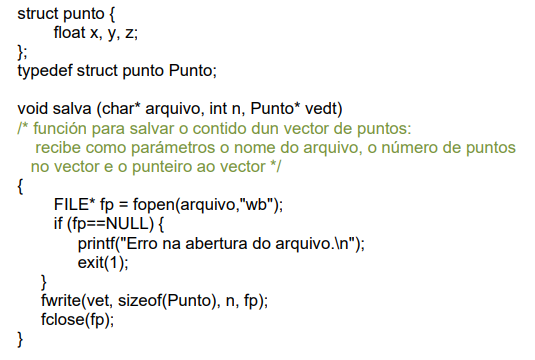
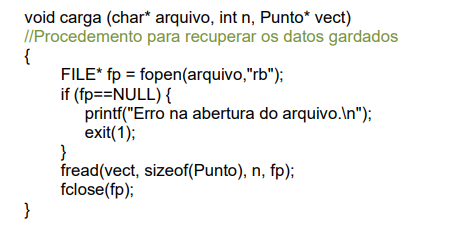


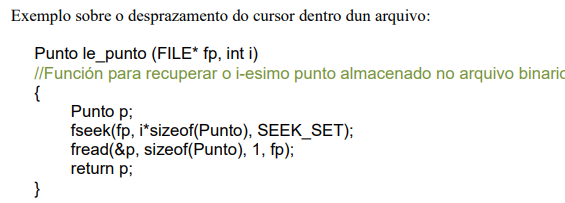
**Arquivos de acceso directo** (binario)

* Aqueles nos que é posible acceder a calqueira rexisto especificando a súa posición respecto da orixe do ficheiro.
* Gárdanse os datos sen formato, almacenando directamente a codificación binaria.
  + Por isto, os datos almacenados ocupan os mesmos bits de memoria.
  + Non son lexibles cun editor de texto.

**Funcións para arquivos de acceso directo**

* **int fwrite (void\* p, int tam, int nelem, FILE\* fp);** 
  + Grava datos nun arquivo birnario
  + Parámetros: Enderezo de memoria a partir do cal se van copiar datos, tamaño en bites de cada elemento, número de elementos a gravar, punteiro ao arquivo.
  + Devolve o número de elementos escritos.
* **int fread (void\* p, int tam, int nelem, FILE\* fp);** 
  + Recupera datos dun arquivo binario
  + Parámetros: Enderezo de memoria a partir do cal se van a a copiar os elementos lidos, tamaño de cada elemento, número de elementos, punteiro ao arquivo.
  + Devolve o número de elementos lidos.
* **int fseek (FILE\* fp, long offset, int orixe);** 
  + Cambia a posición do cursor dun arquivo.
  + Parámetros: Punteiro ao arquivo, posición en bytes que se debe desprazar e orixe dende onde empezar a contar:
    - SEEK\_CUR (posición actual), SEEK\_SET(inicio) ou SEEK\_END(final)
  + Devolve 0 se se rematou con éxito.





1. **Variable estática:** Variable declarada no corpo dunha función, almacenadas na memoria estática. Empréganse cando é preciso manter o valor dunha variable entre varias execucións dun módulo. [↑](#footnote-ref-0)