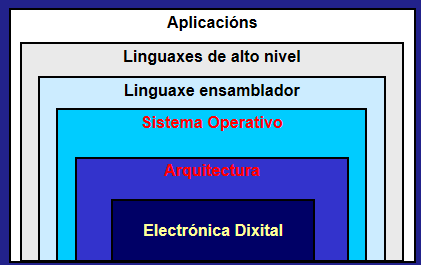
**Algoritmos e programas**

**Visión externa dun ordenador**

* **Ordenadores:** Máquinas de procesamiento automático xeral de datos para a obtención de información.
  + Pódeselles subministrar instrucións para ser executadas sobre un conxunto de datos de entrada e que nos devolva un produto (información / decisión / comando de saída)



**Programas**

* **Procesos necesarios para a creación dun programa:**
  + Especificación e análise do problema
  + Deseño do algoritmo
  + Escritura do programa
  + Validación do programa
  + Mantemento
* **Algoritmo:** Secuencia finita, precisa e ordeada de accións.
* **Programa:** Expresión dun algoritmo nunha linguaxe artificial formalizada.

**Linguaxes de programación**

* Nunha linguaxe de programación:
  + Léxico: Conxunto de símbolos que se poden empregar
  + Sintaxe: Especifica como se poden construír instrucións mediante o léxico
  + Semántica: Asigna un significado a cada tipo de construcción permitida pola sintaxe.
* Tipos de linguaxes:
  + **Linguaxes máquina:** Interpretadas directamente polo procesador, binario
  + **Linguaxes de baixo nivel**: Primeiro nivel de abstracción, dependen da máquina, mnemotécnicos.
    - Convértense a código máquina mediante un ensamblador
  + **Linguaxes de alto nivel**: Empregan palabras comprensible e notación matemática, facilitando a programación.
* **Compilador:** Le código fonte (alto/baixo nivel) e tradúceo a binario, xerando un ficheiro binario executable.
  + Executar programas xa compilados é máis rápido.
* **Intérprete:** Le código fonte e tradúceo a binario e ademais execútao.
  + Para isto, gárdase o código binario na RAM e logo pásase á CPU.
  + Útil para a depuración de programas, permite que se execute en diferentes plataformas (por exemplo na web)
* **Enlazador:** Inclúe tras a compilación bibliotecas estáticas no código, que se empregan no inicio da execución.**Erros**
* Funcionamento inadecuado dun programa. Poden impedir a traducción do código ou abortalo/producir resultados erróneos (bug)
  + Erros de sintaxe > Erros de execución > Erros lóxicos (os útlimos 2 requiren depuración)

**Especificación**

* Pódese especificar o funcionamento dun algoritmo comparando o seu estado inicial (condicións das variables iniciais) co estado final (condicións das variables finais)
* Un algoritmo A verifica unha especificación con precondición P e poscondición Q **({P} A {Q})** se partindo nun estado inicial no que P é verdadeiro, execútase A e no estado final Q é verdadeiro.
  + Exemplo (algoritmo que calcula raiz cadrada das coordenadas dun vector)
    - **Precondición:**

{(∀i: 0<=i AND i<N: v[i]>0) AND (∀k: 0<=k AND k<N: v[k] == Vk.Vk) }

(Toda coordenada de v (vector que ten N coordenadas) debe ser mayor que 0), e exprésase cada coordenada como o producto de dous números

* + - **Poscondición:**

{∀i: 0<=i AND i<N: v[i]==Vk}

(Toda coordenada de V exprésase como un dos números que antes fora multiplicado por si mesmo, é dicir, é a raíz cadrada do valor orixinal)

* **Estado:** Situación na que se atopa unha cousa nos aspectos que son susceptibles de cambiar
  + Definirse mediante conxuntos de pares (variable e valor)
* **Proposición:** Expresión á que se lle pode atribuír un valor de ‘VERDADEIRO’ ou ‘FALSO’
  + Pódense combinar mediante operadores lóxicos, formando novas proposicións

**Predicado**

* Xeralización dunha proposición empregando identificadores, proposicións, operadores e cuantificadores:
  + **∃** *i*: dominio: p(i): existe un ‘i’ no dominio que cumpra ‘p(i)’
  + **∀** i : dominio : p(i): todo ‘i’ no dominio cumpre p(i)
  + **N** i : dominio : p(i): existe un número N de valores i que cumpren p(i)

**Regras de consecuencicia**

* 1ª: Se a especificación {P} S {Q} é correcta e R=>P (R é máis forte (restritiva) ca P), tamén é correcta a especificación {R} S {Q}
* 2ª: Se é correcta a especificación {P} S {Q} e Q=>T (Q é máis forte que T), tamén é correcta {P} S {T}

**Variables e instrucións**

**Datos**

* Os datos son elementos que serven de base para o exacto coñecemento dunha cousa.
* Son os valores iniciais para a solución dun problema mediante un algoritmo, e tamén os resultados do procesamento realizado.
* Cada dato está almacenado nun enderezo na memoria (ex: 0x0022FF24) que se asocia co nome da variable.

**Constante** (#define identificador valor)

* Valor que non pode cambiar durante a execución do programa
* O valor pode ser unha constante ou unha expresión constante (n\*10)

**Tipo de datos**

* O tipo dun dato determina:
  + O rango de valores que pode tomar
  + A súa representación interna
  + O conxunto de operacións que se poden realizar con eles
* A asignación de tipos aos datos ten varios obxectivos:
  + Indicar a representación axeitada na memoria
  + Detectar erros de operacións
  + Determinar como executar as operacións
* **Tipos de datos:**
  + Estáticos:
    - Simples: int, float, double, char
    - Compostos: vector, matriz, estrutura
  + Dinámicos: punteiros

**Int**

* Valores numéricos positivos e negativos (enteiros)
* Limitados a un rango simétrico en torno a 0 que depende dos bits de memoria. (2^(n-1) onde n son os bits)
  + n habitualmente é 16 ou 32. Pódese coñecer cal é mediante sizeof(int).
* En binario, pódese representar como ‘signo x magnitude’ (o primeiro bit indica signo’, pero en realidade empreganse variantes como o ‘complemento a 2’ (o negativo é o contrario do positivo e súmase 1)
* Operacións asociadas: +, -, \*, /, %(resto). O valor do resultado sempre é enteiro.

**float, double, long double** (reais)

* Valores reais positivos e negativos, precisión limitada (double posúe o doble)
  + O rango e a precisión dependen do ordenador
  + Debido á imprecisión, pode ocorrer que (a/b)\*b != a
* Require aproximacións para numeros irracionais
* Representación (en 64 bits): Signo x mantisa x Base^exponente
  + Signo: 1bit
  + Exponente: 11 bits (codificación en exceso, o valor réstase a unha constante)
  + Mantisa: 53 bits
* Operacións: +, -, \*, /

**char**

* Calqueira carácter disponible no teclado dun ordenador, alfabético / numérico / puntuación / control
* A maioría dos ordenadores adoptan a táboa ASCII
  + Asociados cun valor binario de 1 byte (-128,127 se signed ou 0,255 se unsigned)
  + Se definimos char c = ‘a’ e realizamos printf("%d, %c", c, c); imprime ‘97, a’.

**Booleano**

* Poden ser TRUE ou FALSE
* Non existen en C, o resultado de operacións lóxicas é 0 ou 1.
* **Operadores lóxicos:** Permiten combinar operaci´pns relacionais dando lugar a predicados máis complexos.
  + &&(AND), ||(OR), !(NOT)
  + Se ao empregar un && a primeira condición é falsa, non se evalúa a segunda.

**Variable**

* Espazo na memoria asociado a un nome que permite almacenar, ler e modificar un certo dato.
  + Modifícanse mediante ‘sentencias de asignación’ (variable = expresión)
* Requiren un identificador e un **tipo** de dato.
* O espazo de memoria noméase con & e o nome da variable. Para leer este enderezo, úsase scanf(%tipo, &variable);

**Compatibilidade de tipos**

* Ao empregar un dos operandos que se aplican a máis dun tipo de variable (+, \*, etc), se se intenta operar entre variables de distintos tipos, ocorrerá unha conversión:
  + Se algún operando é un long double, o outro convértese a long double
  + Se non, se algún é un double, o outro convértese a double
  + Se non, se algún é un float, o outro convértese a float
  + Se non se cumple ningunha, ambos se toman como enteiros. Logo, realízase a mesma cadea de prioridade con unsigned long int > long int > unsigned int > int.

**Iniciacións**

* Ao iniciar unha variable declárase obligatoriamente o seu nome e tipo.
* Opcionalmente, pódese aportar tamén o seu valor inicial: int a=0.
* Pódese empregar tamén int b=a. Para isto a debe ter un valor asignado.

**Instrucións de entrada/saída**

* **Saída:** Procedemento printf
  + Permite imprimir texto xunto con variables mediante a especificación de formato %c, %d, %u(unsigned int), %f, %e(double ou float en formato científico), %g(double ou float no formato máis apropiado), %s
  + Tamén permite caracteres de escape: \n, \t(tab), \r(retroceso)
  + Para escribir os caracteres “ ou \, escribense con \” e \\.
* **Entrada:** Procedemento scanf
  + Funciona igual ca o printf.
    - A especificación de formato é distinta no caso dos float, que se leen con %f, %e ou %g. Os double lense con %lf, %le e %lg.

**Cadeas de carácteres**

* En C decláranse na mesma forma ca outro vector, pero de caracteres.
* **char** nome\_cadea [tamaño].
* Os strings rematan sempre cun carácter nulo, \0, correspondente á posicion 0 da táboa ASCII.
  + Para declarar unha string por declaración débese incluír este caracter na declaración: **char** cidade[] = {‘V’, ‘i’, ‘g’, ‘o’, ‘\0’};
  + Non se pode asignar unha string enteira fóra da súa declaración. É preciso asignar cada caracter por separado, por exemplo, mediante un bucle for.
* Ao empregar **scanf** co especificador %s, só se le a primeira palabra, pois scanf remata ao ler un espazo.
  + Para solucionar isto, scanf("%[^’\n’]", cidade) le ata o \n.
    - nótese que non se emprega &, ao ser un vector.
  + Tamén é posible empregar o procedemento **gets:** gets(cidade)

**Programación estruturada**

**Control secuencial**

* As instrucións execútanse na mesma orde na que están escritas.

**Estructuras de control**

* É posible construír calquera programa mediante a combinación dun conxunto reducido de estruturas de control secuenciais, selectivas e repetitivas.
  + **Selectiva:** Úsase para tomar decisións lóxicas. If, if else, switch.
    - Switch case permite intervalos (case a… b)
    - En C, a condición pode ser unha expresión lóxics ou unha variable enteria (0 equiválese co falso)
  + **Repetitivas:** Realizan varias veces o mesmo conxunto de operacións. for, while, do-while
    - O for utilízase cando se coñece o número de repeticións

[ for (iniciación; condición; incremento) ]

* + - Repíteme mentres que a condición sexa certa.
  + Operadores:
    - De comparación: <, >, ==, <=, >=, !=
    - Lóxicos: &&, ||, !
    - Preferencia operadores: ! e funcións, aritmético, de comparación, lóxicos, ?:, de asignación (=, +=, -=, /=, \*=)

**if, else if**

* Execútase se se cumpre a condición especificada.
* Pode ser seguido dunha cantidade indefinida de else if, e pode rematar con un ‘else’ para o caso xeral. Despois do ‘else’ non pode haber outro else if.

**switch(n)**

* Composto por varios apartados **case** valor:, que se executan cando o valor correspóndese co da variable.
* No final posúe un caso **default:** que se executa cando non se cumple ningún dos previos.
* Todos os case deben rematar cun **break**. Do contrario, ao analizar un caso que non ten break, pasa ao seguinte sen considerar a súa condición. O default non precisa break, e cando un case contén só un comando e un break non require {}.

**For**

* Execútase ata que unha variable, que cambia con cada iteración, chega a certo valor.
* Útil para cando o valor de repeticións é coñecido, como ao editar vectores.
* Para matrices, empréganse 2 for aniñados.

**While**

* Execútase ata cumplir unha condición. Se se emprega un do{}while(); executarase unha vez primeiro e despois comprobará a condición.
* Dentro dun bucle while, o predicado que se mantén constante dende inicio a fin chámase **invariante.**
* Por exemplo, nun programa que procura a primeira aparición de x nun vector A[N]: {0 <= i <= N AND x ∉ {A[0],..., A[i-1]}} é o invariante, mentres que {0<= i <= N AND x ∉ {A[0],..., A[i-1]} AND (i== N OR x== A[i]) é a poscondición.
* É o máis axeitado para buscar un elemento nunha lista, pois pode deterse cando o atopa.

**Vectores**

* Declaranse mediante: int v[5], float a1[]={2.0, 4.5, 6.2}
* Operacións: a3[1] = a2[1] + 2\*a1[1]. Non se pode operar entre vectores, só elemento a elemento.
* Só se poden asignar valores a un vector cando se declara inicialmente.
* Na memoria almacénanse en posicións consecutivas na memoria. Un vector de n enteiros ocupa n\*sizeof(int).

**Matrices**

* float mat2[10][24]: define unha matriz 10x24. Os índices comezan en 0
* Os datos almacénanse en memoria na orde de leer a matriz de esquerda a dereita e de arriba abaixo.
  + O espazo no que se almacena é:

(espazo do primeiro elemento + i\*n\_cols + j)\*sizeof(float)

**typedef** tipo\_elemento

* Permite definir un novo tipo de dato. Exemplo: **typedef** tipo\_elemento tipo\_matriz [N][M] (sendo tipo\_elemento o tipo de dato na matriz, por exemplo float ou int)
  + Por exemplo: **typedef** float Matriz10x10[10][10]; Matriz10x10 mat1;
  + define mat1 como unha matriz 10x10.

**Rexistro de datos**

* Represéntase mediante struct nome {tipo\_dato dato1; tipo\_dato dato2;...}
* Emprégase para crear unha estrutura con varios campos de datos.
* Accédese ás variables mediante nome.dato1
* Ao empregalas como parámetro nun procedemento é preciso empregar **struct** e o nome da estructura.
* Un **rexistro xerárquico** é aquel que contén un rexistro como variable.

**Acceso secuencial**

* Acceso á memoria mediante bucles for. Por exemplo, para modificar todos os valores dun vector ou dunha matriz.

**Deseño descendente**

* Estratexia de programación de problemas non triviais.
  + Descompor o problema en subproblemas máis sinxelos
  + Descompor cada subproblema para chegar a accións implementables de forma directa.
  + Exemplo: Considerar variables necesarias, ler o seu valor, comprobar se son válidas, realizar a operación
* Para un esquema repetitivo, identificamos as accións útiles a repetir e as variables necesarias, identificamos como actualizar a información entre iteracións, logo a condición de terminación e finalmente os valores iniciais das variables.
* **Diagrama de fluxo de datos:** Esquema gráfico que expresa o proceso de deseño descendente.
  + Accións expresadas de xeito iterativo e incremental
  + Identifícanse os datos e variables necesarias
  + Especifícanse os entes externos cos que interactúa o programa
  + Formado por un diagrama de contexto (datos de entrada e saída) e varios DFDs en niveis interemedios.
    - Nun esquema, os datos non poden ir entre entidade externa e almacén directamente, debe pasar todo por un proceso
  + **Convencións:**
    - Nomes significativos para procesos, fluxos e almacéns
    - Numerar os procesos cun único número na parte superior.
    - Para precisar máis, pódense expandir en novas páxinas para outros niveis de DFD

**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.

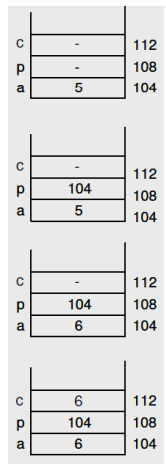
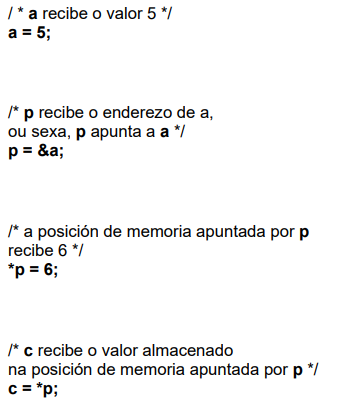
**Uso de memoria**

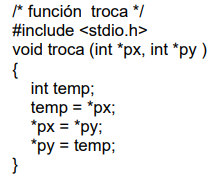
* Durante a compilación non é posible determinar todos os requerimentos de memoria. Distinguimos entre 3 partes da memoria do ordenador:
  + **Memoria estática:** Empregado para variables globais, estáticas[[1]](#footnote-0) e instrucións. Asignada durante compilación.
  + **Memoria dinámica:** Empregado 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.

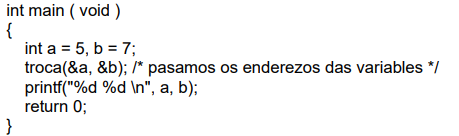
**Paso de argumentos**

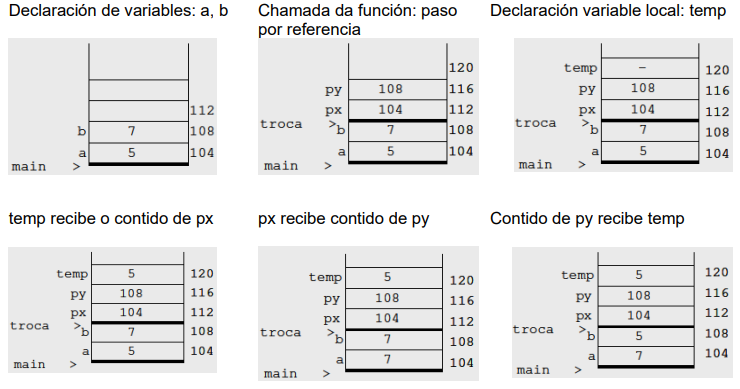
* Comunican os módulos entre si.
* **Parámetros formais:** Sitúanse na definición do módulo.
* **Parámetros actuais:** Sitúanse na invocación do módulo.
* Diferenciamos entre parámetros de entrada, saída e entrada/saída (modifícanse na función).
* Todos se poden empregar e pasar por valor ou por referencia.
  + 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.

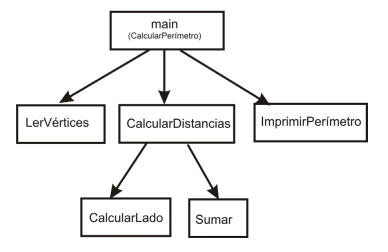
**Rexistros como parámetros**

* 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

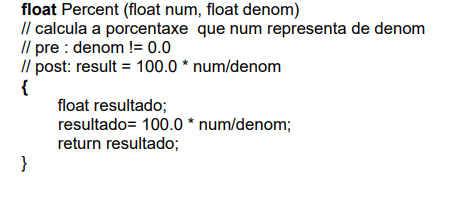
**Activación dun subprograma**

* 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.

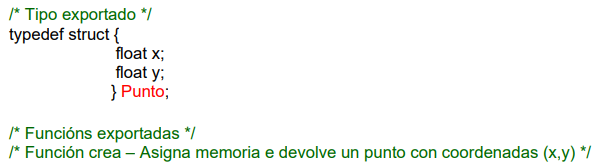
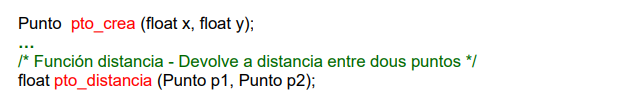
**Programación por contrato**

* 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.

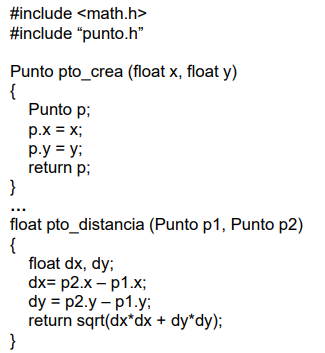
**Bibliotecas**

* 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:
  + 
  + 

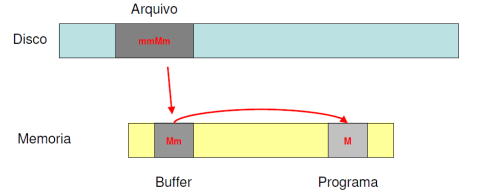
**Implementación dun módulo**

* 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.

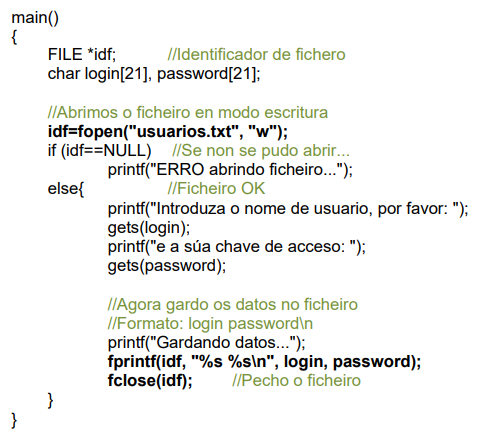
**Arquivos de acceso secuencial**

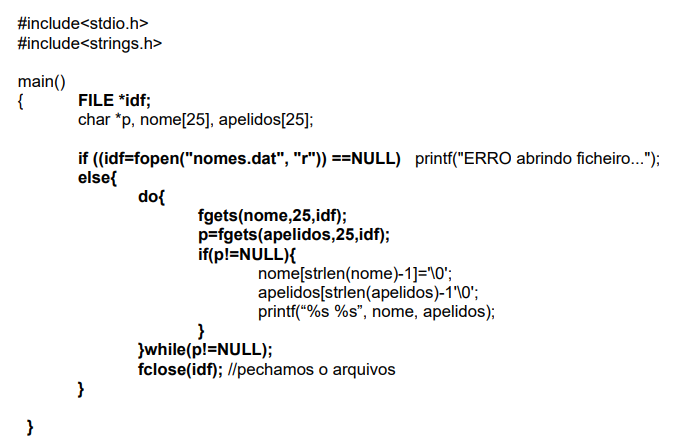
* Nun **ficheiro de acceso secuencial** o acceso a un dato esixe o acceso previo a todos os datos interiores.
  + Modos diferentes para escritura e lectura
    - No modo escritura o ficheiro gárdase dende 0, sobreescribindo os datos previos se non se volven a introducir.
    - Nalgúns casos permítese 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.

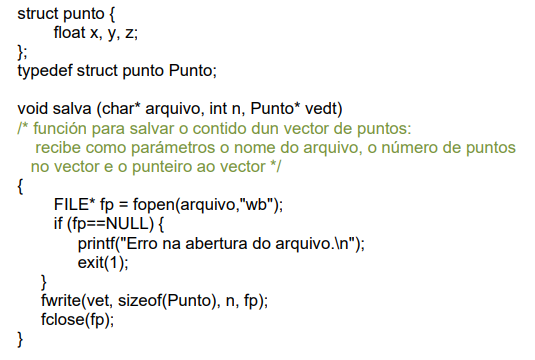
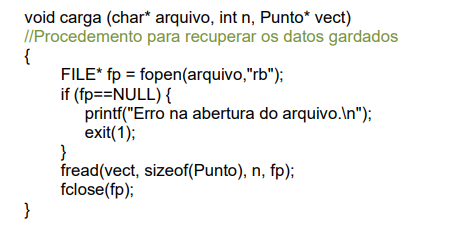


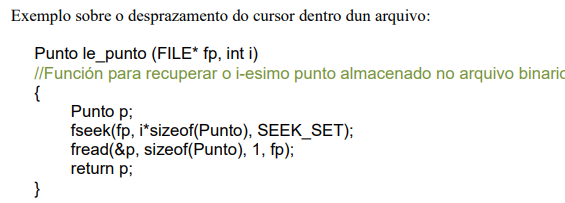
**Arquivos de acceso directo** (binario)

* 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 binario
  + 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)