Modularizált programon azt értjük, amely több, jól körülhatárolható részfeladat megoldásaiból épül fel. Egy-egy részfeladat gyakran szabványos módon, a programozásban jól ismert programozási minták, az úgynevezett programozási tételek alapján oldható meg. Összetett programkódokat célszerű a megoldandó részfeladatok szerint csoportosítani. Ezek a csoportok többnyire fizikailag is elkülönülnek a programkódban, ezáltal áttekinthetőbbé teszik a megoldást. Ebből a célból programkódunkban létrehozhatunk egyszerű utasítás blokkokat, alprogramokat (függvényeket, eljárásokat) illetve több önálló fordítási egységet.

Alprogramokra tagolt program szerkezete

Egy alprogram a funkcionálisan összetartozó utasításokat, azaz egy-egy részfeladat megoldására képes program-kódot tartalmaz. Ezen frappáns megfogalmazás ellenére az egyik legnehezebb kérdés, hogy a program mely részeit valósítsuk meg külön alprogramban.

Ha ugyanaz a kódrészlet programunk több helyén is megjelenik, akkor célszerű azt egy külön alprogramban elhelyezni, és ahol szükség van rá, meghívni. Így egyrészt kevesebbet kell írni, másrészt ha javítani kell a kérdéses kódrészleten, akkor azt csak egyetlen helyen kell megtenni.

A másik általános elv az, hogy egy alprogram kódját egyszerre lássuk a képernyőn. Ha ez nem állna fenn, akkor tagoljuk a kódot részekre, a részeket csomagoljuk külön alprogramokba. A tagolásnál azt is tartsuk szem előtt, hogy egy alprogramban ne szerepeljen egy-két ciklusnál több, továbbá ha a ciklus magja túl nagy, akkor azt is tagoljuk.

A programozási tételek segítségével megoldott feladatoknál általában egy programozási tételnek megfelelő programkód alkot egy önmagában összetartozó részt. Ilyenkor kézenfekvő a kódot a programozási tételek mentén csoportosítani. Így egy-egy alprogramba egy-egy tétel kódrésze kerül. A programozási tételek többnyire egyetlen ciklust tartalmaznak. (Kivétel például egy mátrixban történő számlálás, ahol egy dupla `for` ciklus található. Természetesen ilyenkor nem szedjük szét két egymásba ágyazott alprogramra a programot.) Ha egy programozási tételbe beágyazunk egy másik tételt, akkor a külső programozási tételt tartalmazó alprogram fogja meghívni a belső programozási tételt tartalmazó alprogramot. A külső programozási tétel állhat közvetlenül a főprogramban, de lehet külön alprogram is.

Gyakori, hogy az adatbeolvasást, az adatkiírás vagy azok egyes részeit azért helyezzük el önálló egységekbe, hogy a főprogramban az amúgy terjedelmes input-output részek ne foglaljanak el túl sok helyet.

Nagyobb programok készítésekor célszerű a rokon alprogramokat csoportosítani, és a csoportokat külön-külön fordítási egységbe szervezni.

C++ nyelvű kódban egy csoportot mindig egy fej- és forrásállomány párban helyezünk el: a `h` kiterjesztésű (például `modul.h`) fejállomány a függvények deklarációit tartalmazza, a `cpp` kiterjesztésű (például `modul.cpp`) forrásállomány pedig a függvények definícióit. A fejállományt mind a `modul.cpp` forrásállományba, mind az összes olyan állományba be kell „inklúdolni” (`#include "modul.h"`), ahol a felsorolt függvényeket használni akarjuk. A függvény deklarációk előtt adjuk meg a fordítási direktívákat (például a `#include` sorok), és a `using namespace std` utasítást. A függvény deklarációk előtt állhatnak még típus-definíciók, konstans definíciók (globális változó definíciókat ne használjunk).

Egyetlen olyan forrásállományunk lesz csak, amelyhez nem tartozik fejállomány: ez a `main()` függvényt is tartalmazó állomány. A `main()` függvény egy speciális függvény, amelynek hívását az operációs rendszer kezdeményezi, amikor a programunkat futtatjuk. Ha

ebben az állományban a `main()` függvényen kívül más függvények is vannak, akkor azokat a `main()` függvény után definiáljuk, de deklarációjukat a `main()` függvény előtt helyezzük el.

Alprogramokra tagolt program végrehajtása

Az alprogram egy olyan programblokk, melynek végrehajtását a program bármelyik olyan helyéről lehet kezdeményezni, ahol az alprogram neve érvényes (ahová az alprogram hatásköre kiterjed). Amikor az alprogramot a nevére történő hivatkozással „meghívjuk”, akkor a vezérlés (az utasítások végrehajtásának menete) átadódik a hívás helyéről az alprogram első utasítására. A hívó program további végrehajtása mindaddig szünetel, amíg a függvényhez tartozó kód le nem fut. Az alprogram akkor fejeződik be, ha vagy az összes utasítása végrehajtódott, vagy egy befejezését előíró (`return`) utasításhoz nem jut. Ekkor a program végrehajtása visszakerül a hívás helyére.

Az alprogramok többféleképpen bonyolíthatnak le adatforgalmat az őket hívó programrésszel: használhatnak közös adatokat (ezt csak nagyon indokolt esetben használjuk), kiinduló értékeket kaphatnak a hívás helyéről paramétereken keresztül, és értékeket adhatnak vissza a hívás helyére vagy paramétereken keresztül vagy speciális visszatérési értékként.

Paraméterek és a visszatérési érték

Az alprogramokba csomagolt kódrészek egy-egy részfeladatot oldanak meg. Ennek a részfeladatnak ugyanúgy vannak bemenő adatai és eredményei, mint az eredeti feladatnak azzal a különbséggel, hogy ezeket az adatokat általában a hívás helyéről kapja az alprogram, és a hívás helyére adja vissza.

A hívás helyével folytatott adatcserét paraméterváltozók segítségével bonyolítjuk le. Az alprogram deklarációjakor az alprogram neve utáni zárójelek között soroljuk fel a formális paraméterváltozókat (típusukkal együtt). Külön meg kell jelölni, hogy mely változók kötődnek csak bemenő, és melyek kimenő adatokhoz.

A alprogram hívása annak nevével történik. E név után zárójelek között kell felsorolni az aktuális paramétereket: a bemenő adatokat változó vagy kifejezés formájában, illetve a visszakapott adatokat tartalmazó változókat. Fontos, hogy a formális és aktuális paraméterek száma, típusaik sorrendje megegyezzen, a hivatkozás szerinti formális paramétereknek megfeleltetett aktuális paraméterek mindig változók legyenek. (Bizonyos esetekben a formális paramétereknek lehetnek alapértelmezett értékeik. Ilyenkor az aktuális paraméter lista természetesen lehet rövidebb a formális paraméterlistánál.)

Minden függvény (C++ nyelvben a nem void típusú függvény) képes egy értéket különleges módon, visszatérési értékként visszaadni a hívás helyére. Ezt az értéket a hívás helyén maga a függvény-kifejezés képviseli, a függvényben pedig a `return` utasítás után kell valamilyen kifejezés formájában leírni. A viszatérési érték típusát a függvény deklarációjakor a függvény neve elé írt típus (a függvény típusa) határozza meg. Ez a típus speciális esetben lehet `void` is, ami azt jelenti, hogy a függvénynek nincs visszatérési értéke. Ilyenkor a `return` utasítás után nem áll kifejezés, és a függvény hívása önálló utasításként (pontosvesszővel lezárva) jelenik meg, nem pedig egy kifejezésben.

Ha egy függvénynek van visszatérési értéke, akkor lehetőleg ne legyenek kimenő adatokat hordozó paraméterei, azaz ilyenkor a paraméterek csak a bemenő értékeket közvetítsék. Ettől a szabálytól indokolt esetben el lehet térni. Ilyen indokolt esetek például a programozási tételeket beágyazó függvények.

Programozási tételek függvényei C++ nyelvben

A számlálás függvényére egyetlen kézenfekvő megoldás kínálkozik:

```
int szamlal(const int t[], const int n)
{
   int s;
   ...
   return s;
}
```

A feltételes maximum keresést többféleképpen is definiálhatjuk.

1. Mind a bemenő, mind a kimenő adatok a paraméter listán szerepelnek. A bemenő adatok (általában) konstansok, a kimenő adatok változóit elemi típusok esetén megelőzi az `&` jel.

```
void  feltmaxker(const int t[], const int n,
                 bool &l, int &max, int &ind)
{
   ...
}
```

2. A kimenő adatok visszatérési értékként kerülnek ki a függvényből. Mivel több kimenő adat is van, ezeket egy struktúrába kell összefogni. A `result` egy rekord szerkezetű típust jelöl, amelynek három adattagja (mezője) van. Egy `r result` típusú változónak külön-külön hivatkozhatunk a tagjaira: `r.l`, `r.max`, `r.ind`.

```
struct result { bool l; int max; int ind;};
result  feltmaxker(const int t[], const int n)
{
   result r;
   ...
   return r;
}
```

3. Az egyik kimenő adatot kiemeljük, és az visszatérési értékként jelenik majd meg a függvényben, a többi paraméterként.

Mi ezt az utolsó változatot használjuk majd.

```
bool feltmaxker(const int t[], const int n, int &max, int &ind)
{
   bool l;
   ...
   return r;
{
```

A lineáris keresést is – hasonlóan a feltételes maximum kereséshez – többféleképpen deklarálhatjuk.

1. `void  linker(const string t[], const int n, bool& l, int &i);`
2. `struct result { bool l; int i;};`
   `result  linker(const string t[], const int n);`
3. `bool  linker(const string t[], const int n, int &i);`

Itt is a harmadik változat tűnik a legjobbnak.

Program modularizálása

Egy összetett program kódja áttekinthetőbbé válik, ha a logikailag összetartozó részeket a kód többi részétől elkülönítve, külön állományban, modulban tároljuk. A fizikai elkülönítés erősíti az elkülönített részen belüli összetartozást. Összegyűjthetjük a hasonló célú (feladatú) és tartalmú függvényeket (például tömbökkel kapcsolatos beolvasásokat , esetleg bonyolult matematikai számításokat végző eljárásokat). Ezt nevezzük procedurális modularizációnak. Amikor egy felhasználói típus megvalósításának elemeit (az osztály és metódusainak definícióit) különítjük el, akkor típus orientált (objektum orientált) modularizációról beszélünk.

A különválasztással a programunk külön részekben is fejleszthető (fordítható), így lehetőség nyílik a program csoport munkában történő elkészítésére.

További előny, hogy az önállóan fejlesztett, lefordított, kipróbált programrészt (komponenst) később más programokba is beilleszthetjük, azaz újrafelhasználhatjuk.

A modulokra széttagolt programoknál ügyeljünk a kód egyes részei közötti kapcsolat kialakítására. Pontosan kell meghatározni, melyek azok a függvények, felhasználói típusok, amelyeket az egyik modulban definiálunk, de egy másik modulban akarjuk felhasználni.

C++ nyelvben a fő forrásállomány tartalmazza a `main()` függvényt. E mellett számos más függvény is lehet. Ezeket a `main()` függvény után definiáljuk, deklarációjukat pedig a `main()` függvény előtt helyezzük el. A teljes kód egy részét azonban a `main()` függvényt tartalmazó forrásállományon kívül más önálló fordítási egységként kezelt állományokban is elhelyezhetjük. Ezeket az kódrészeket mindig két állományba szedjük szét: egy fejállományba és egy forrásállományba. A fejállományba azon elemek (osztály definíciók, függvény deklarációk) kerülnek, amelyeket más állományokban szeretnénk használni, minden egyéb a forrásállományba kerül. Célszerű az összetartozó fej- és forrásállományokat azonos névvel, de eltérő kiterjesztéssel ellátni. A fejállomány kiterjesztése `.h` , a forrásállományé `.cpp`. A fejállományt mindig be kell illeszteni (`#include`) a hozzátartozó forrásállományba. A forrásállomány rendszerint függvény- illetve metódus-definíciókat tartalmaz. Megjelenhetnek benne belső, tehát saját típusok, konstansok és függvények is. A forrásállomány elején helyezzük el a típus-definíciókat, a konstans-definíciókat, és a belső függvények deklarációit. Ezt követik a függvény- illetve metódus-definíciók.

Azokban az állományokban (a fej- és forrásállományaiban egyaránt), ahol szükségünk van egy másik fejállományában leírt elemekre, a szolgáltató fejállományát be kell illeszteni (`#include`) a szolgáltatást igénybe vevő állományba.

Kódolási megállapodások

1. Alprogramba tegyük be azokat a kódrészeket, amelyeket a programunk több különböző helyén is használni akarunk.
2. Tördeljük a programkódunkat alprogramokba úgy, hogy egy-egy alprogram egy-egy részfeladat megoldásáért feleljen.
3. Az egyes programozási tételeket önálló kódrészként (blokk, alprogram) helyezzük el.
4. Az összetett beolvasó illetve kiíró részeket is alprogramokban helyezzük el.
5. Az alprogram paramétereit az adott programrész bemenő és kimenő adatai képezik. Egyetlen kimenő adat esetén annak értékét a függvényként megadott alprogram visszatérési értéke hordozza. Több kimenő adatnál választhatunk: egy összetett szerkezetű visszatérési értéket használunk vagy több kimenő paramétert. Kerüljük a kevert

megoldást! Vegyes formát olyan programozási tételek kódolásánál indokolt alkalmazni, ahol a logikai érték egy kitüntetett szerepű kimenő adat.

6. A függvény (C++ nyelvben a nem void típusú függvény) végrehajtása mindig `return` utasítással érjen véget.
7. Külső, azaz több forrásállományra nézve globális változót ne használjunk.
8. Adott forrásállományra nézve globális változókat csak indokolt esetben alkalmazzunk, akkor is csak keveset. (Például ha egy változó majdnem mindegyik függvényben szerepel, ezek közül csak egyben változik az értéke, a többiben pedig csak lekérdezzük azt.)
9. C++ : A fejállományokat úgy tesszük „egyszer betölthetővé”, hogy az első sorába az `#ifndef NEV_H` utasítást (a `NEV_H` névben a `NEV` a fejállomány fizikai neve kiterjesztés nélkül), a második sorba a `#define NEV_H` utasítást, és az utolsó sorba a `#endif` utasítást írjuk.
10. C++ : Amennyiben egy fejállományban hivatkozunk a `string`, `ifstream` vagy `ofstream` típusra, akkor azok elé oda kell írni az `std::` minősítést.
11. C++ : Minden forrásállományba illesszük be a `using namespace std` sort.