Egyszerű programon azt a konzolos (a felhasználóval egy konzol-ablakon keresztül kommunikáló) alkalmazást értjük, amely utasításai egyetlen forrásállományban, egy vezérlési láncra vannak felfűzve, kódja nincsen függvényekre, eljárásokra tagolva. Az alább ismertetett implementációs tanácsok természetesen az ennél összetettebb programokra is érvényesek.

Program szerkesztése, fordítása, tesztelése

Alkalmazásunkat minden esetben egy úgynevezett projektbe ágyazzuk. Eleinte, amíg az alkalmazásunk egyetlen forrás állományból áll, talán körülményesnek tűnik ez a lépés, de ha megszokjuk, akkor később, a több állományra tördelt alkalmazások készítése sem lesz nehéz.

A bizonyítottan helyes absztrakt program kódolása többnyire nem eredményez működő programot. A kódolás során ugyanis elkövethetünk alaki (szintaktikai) hibákat (ezt ellenőrzi fordításkor a fordító program) és tartalmi (szemantikai) hibákat. Ez utóbbiak egy részét a kódolási megállapodások betartásával tudjuk kivédeni, más részét teszteléssel felfedezni, a hiba okát pedig nyomkövetéssel megtalálni. A programok kódolásánál törekedjünk arra, hogy a beírt kódot olyan hamar ellenőrizzük fordítás és futtatás segítségével, amilyen hamar csak lehet. Ezért ne a teljes kód beírása után fordítsuk le először a programot, mert a hibaüzenetek tényleges okát ekkor már sokkal nehezebb megtalálni. Eleinte ne szégyelljük, ha utasításonként fordítunk, de később se írjunk le egy program-blokknál többet fordítás nélkül. Ezzel a szerkesztés során elkövetett szintaktikai hibákat viszonylag könnyen ki tudjuk szűrni, hiszen egyszerre csak kis mennyiségű kód javításával kell számolnunk.

A fordításnál keletkezett hibaüzenetek értelmezése nem könnyű. A fordítóprogram annál a sornál jelez hibát, ahol a hibát észlelte, de a hiba gyakran több sorral előbb található vagy éppen nem található (például egy változó deklarálásának hiánya). Érdemes a hibaüzeneteket sorban értelmezni és kijavítani, mert bizonyos hibák egy korábbi hibának a következményei. Fontos tudni, hogy a hibaüzenet nem feltétlenül adja meg a hiba javításának módját és helyét. Ezért nem szabad a hibaüzenet által sugallt javítást azonnal végrehajtani, hanem először rá kell jönnünk a hiba valódi okára, majd meg kell keresnünk, hol és mi módon korrigálhatjuk azt. Ne csak a fordító hibaüzeneteit (error), hanem a figyelmeztetéseit (warning) is olvassuk el. A figyelmeztetések rámutathatnak más, nehezen értelmezhető hibaüzenetek okaira, vagy később fellépő rejtett hibákra.

Érdemes a kódot úgy elkészíteni, hogy annak bizonyos részei minél hamarabb kipróbálhatók, azaz futtathatók legyenek. Így a programot már akkor ki tudjuk próbálni, amikor a teljes feladatot még nem oldja meg.

Ne sajnáljuk a tesztelésre szánt időt. Gondoljuk át milyen tesztesetekre (jellegzetes bemenő adatokra) érdemes kipróbálni a programunkat. A fekete doboz teszteseteket a feladat szempontjából lényeges vagy extrém bemenő adatok adják. A fehér doboz teszteseteket a programkód ismeretében állítjuk elő. Itt olyan adatok megadása a cél, amelyekkel a program minden egyes utasítása legalább egyszer végrehajtódik, továbbá az elágazási és gyűjtő pontjain mindenféle irányból keresztül megy a vezérlés. A tesztelést (akárcsak a fordítást, futtatást) menet közben, már egy-egy részprogramra érdemes végrehajtani. Ha a programunk egy tesztadatra rosszul működik, akkor a hiba kijavítása után újra az összes tesztadatra meg kell ismételni a tesztelést, nemcsak a rossz működést előidézőre.

A nyomkövetés egy tartalmi hiba bekövetkezésének pontos helyét segít kideríteni, noha ez nem feltétlenül az a hely, ahol programot javítani kell. A nyomkövetés a program lépésenkénti végrehajtása, melynek során megvizsgálhatjuk változóink pillanatnyi értékét.

Egyszerű programok implementálása

Egy program kódolása előtt ismernünk kell az általa megoldandó feladatot, annak adatait, az adatok típusát, változóik nevét, azt, hogy melyek a bemenő illetve eredményt hordozó adatok, és ezek milyen feltételeknek tesznek eleget, továbbá a megoldó program absztrakt vázát, azaz rendelkeznünk kell a program tervével.

A terv általában nem tér ki arra, hogy a bemenő változók hogyan veszik fel a kezdőértékeiket, illetve az eredmény változók értékeit hogyan tudjuk a felhasználó felé láthatóvá tenni. Ezért egy működő program elkészítése nem pusztán a tervezés során előállt absztrakt program kódolásából áll, hiszen számottevő kódot kell készítenünk egyrészt az adatok beolvasásának megvalósításához, másrészt az eredmény megjelenítéséhez. Ezért nevezzük ezt a folyamatot a terv kódolása helyett a terv implementációjának (megvalósításának).

Az implementálás menete

1. A program kerete

   Ezen azokat a kötelezően megadandó kódsorokat értjük, amelyek közé kell majd a feladat megoldásának kódját írni. Lényeges része ennek a keretnek az a pont, ahonnan a program vezérlése futtatáskor elindul. Ez azt jelenti, hogy a számítógép e pont után elhelyezett utasításokat kezdi el egymás után végrehajtani.
2. Bemenő adatok beolvasása és ellenőrzése

   A beolvasási szakaszban gondoskodnunk kell arról, hogy a bemenő változók megfelelő kezdőértéket kapjanak. A feladat tervéből kiderül, hogy programunknak melyek a bemenő változói, és azok kezdő értékeire milyen feltételnek kell teljesülnie. A bemenő adatok kezdőértékeihez többnyire a szabványos bemenetről vagy egy szöveges állományból nyerjük. A beolvasott adatokra meg kell vizsgálni, teljesítik-e a szükséges előfeltételeket. Ha nem, akkor hiba-jelzést kell adni, és vagy leállítani a programot, vagy újra meg kell kísérelni a beolvasást. (Ezt a kódrészt bizonyos esetekben összevonhatjuk a programnak a számítást végző részével. Ilyenkor az adatokat a számítás menetéhez igazodva, a számítással egy időben olvassuk be.)
3. Az absztrakt program kódolása

   A program második része az absztrakt programnak megfelelő kód. Amennyiben ez strukturált, akkor annak szerkezete alapján kívülről befelé haladó kézi fordítással állítjuk elő a kódot. Mindig az adott szerkezeti elemnek (szekvencia, elágazás, ciklus) megfelelő utasítást kódoljuk. Az elemi utasításokat közvetlenül kódoljuk. A programkódban tabulálással is jelezzük a program szerkezetét.
4. Eredmény megjelenítése

   Az eredmény változók értékének megjelenítése (a szabványos kimeneten vagy egy szöveges állományban) alkotja a programkódunk harmadik, egyben befejező részét. Néha ezt a kódrészt összevonhatjuk az előző, számítást végző résszel, és az eredményt a számítással párhuzamosan írjuk ki.
5. Az alkalmazott változók deklarálása

   A programkódban használt változók típusát meg kell adni, azaz a változókat deklarálni kell. Az állapottér változóit a kód elején deklaráljuk, a segédváltozókat elég abban a blokkban, ahol használjuk őket, természetesen még az első hivatkozás előtt. (Blokknak tekintjük a teljes kódot, de az egyes programszerkezetek által befoglalt kódrészeket is.

Kódolási megállapodások

1. A kódot úgy kell elkészíteni, hogy annak tördelése jól tükrözze a program szerkezetét. Az utasításokat többnyire külön sorba írjuk.
2. Szimultán értékadásokat egyazon sorba írt egyszerű értékadásokkal helyettesítjük.
3. A logikailag összetartozó, például ugyanazon vezérlési szerkezetbe tartozó utasítás-csoportokat (elágazás ágai, ciklusmagja) külön utasítás-blokkba helyezzük. Ez csak akkor mellőzhető, ha az ág vagy a mag egyetlen utasításból áll. Ilyenkor a teljes vezérlési szerkezet egyetlen sorban álljon. Javasolt megoldás: az elágazás ágait és a ciklusmagot akkor is tegyük utasításblokkba, ha az csak egy utasításból áll, így ha újabb utasításokat fűzünk hozzá, nem feledkezhetünk meg a blokk kijelöléséről.
4. A vezérlési szerkezeteket egymástól jól megkülönböztetve kell leírni.
5. A többágú elágazásokat nem több két-ágú elágazás (ahol az „else” ág üres) szekvenciájával, hanem egymásba ágyazott két-ágú elágazásokkal kódoljuk. A sok-ágú elágazás (swtich, case) csak speciális esetben használható.
6. Ciklusok kódolására az elől-tesztelős ciklus-utasítást használunk. Az általános elől-tesztelős ciklus (while) mellett gyakran használjuk az úgynevezett számlálásos (for) ciklust is, ami speciális elől-tesztelős ciklusnak számít. A hátul tesztelős ciklus-utasítást kizárólag az ellenőrzött adatbevitelhez vagy a program ismételt végrehajtásának biztosításához használjuk.
7. Az adatok beolvasása kétféleképpen történhet. Élesen elkülöníthetjük azt a számításokat végző résztől, vagy a számítások közben végzünk folyamatos beolvasást. Ez utóbbi akkor indokolt, ha több azonos típusú érték feldolgozása a feladat, amelyeket nem akarunk egy tömbben tárolni.
8. Egy adat ellenőrzését közvetlenül a beolvasás után kell elvégezni. Ha az adat értéke helytelen, akkor hiba-üzenetet generálunk. Ezt követően kétféle lehetőségünk van: vagy azonnal megállítjuk a program futását, vagy addig olvassuk az ellenőrizendő adatot újra és újra, amíg az helyes nem lesz.
9. A legtöbb fejlesztő környezet automatikusan gondoskodik arról, hogy a program befejeződése után csak egy külön <enter> leütésre szűnjön meg a konzol-ablak. Ha azonban nem egy ilyen fejlesztő környezetben indítjuk a programunkat, akkor annak befejeződésekor a program konzol-ablaka azonnal megszűnik, így nincs lehetőségünk arra, hogy elolvassuk a program futási eredményét. Ennek megakadályozására a program összes megállási pontjára egy várakozó utasítás kell elhelyezni. Az elegáns operációs rendszertől független várakozó utasítás (pl. <enter> leütéséig várakozik) nem mindig van. Univerzális megoldás viszont – bár nem annyira elegáns – egy karakter beolvasására való várakozás.
10. A tömb azonos típusú értékek sorozatának tárolására szolgál. Használata előtt meg kell tudnunk mondani, legfeljebb hány eleme lesz, azaz mekkora memória helyet foglaljunk le számára. Kényelmes használatot biztosít a dinamikus helyfoglalású tömb, amelynek méretét futási időben, de még a használata előtt elég megadni. Találkozhatunk dinamikus (dinamikusan nyújtózkodó) tömbbel is, de ennek használatánál körültekintően kell eljárni, mérlegelve a használatából adódó futási idő növekedést.
11. Szöveges állományok kezelése lehetőséget ad arra, hogy a program bemenő adatait ne közvetlenül a billentyűzetről olvassa, kimenő adatait ne a konzol ablakba írja. A beírandó értékeket előre elhelyezhetjük egy olvasásra szánt szöveges állományba, eredményeket pedig egy másik szöveges állományba írhatjuk. Ilyenkor az előbbi szekvenciális inputfájlként, az utóbbi szekvenciális outputfájlként viselkedik.