F3. A C++ NYELVRŐL C PROGRAMOZÓK SZÁMÁRA

F3.1 A C++ nyelv története

A C++ nyelv kidolgozása az AT&T Bell Laboratóriumoknál dolgozó Bjarne Stroustrop nevéhez fűződik. Mint ahogy ismeretes a C nyelvet szintén itt fejlesztették ki a 70-es évek elején. Így nem kell csodálkozni azon, hogy a 10 évvel későbbi C++ fejlesztés a C nyelvre épült. A C nyelv ismerete ezért teljesen természetes kiindulópont a C++ nyelv megismeréséhez. Bjarne Stroustrop két fő szempontot tartott szem előtt a C++ kidolgozásánál:

1. a C++ nyelv legyen felülről kompatibilis az eredeti C nyelvvel,
2. a C++ nyelv bővítse ki a C nyelvet a Simula 67 nyelvben használt osztályszerkezettel (*class*).

Az osztályszerkezet, amely a C **struct** adatszerkezetre épült, lehetővé tette az objektum-orientált programozás (OOP) megvalósítását.

A C++ nyelv nem szabványosított nyelv. Az C++ Version 1.2 változata terjedt el először a világon (1985). A használata során felvetődött problémák, igények figyelembevételével Bjarne Stroustrop kidolgozta a Version 2.0 nyelvdefiníciót (1988). A jelentős változtatások miatt a régi C++ (1.2) nyelven írt programok általában csak kisebb-nagyobb javítások után fordíthatók le a 2.0-ás verziót megvalósító fordító programmal.

Tudni kell azonban, hogy a C++ nyelvet tovább fejlesztik az alábbi irányokban:
- paraméterezett típusok, osztályok és függvények használata,
- kivételek (*exception*) kezelése.

A Borland C++ 3.x rendszerek szintén a C++ Version 2.0 definíciót használják, kiegészítve a paraméterezett típusok, osztályok és függvények (*template*) definíciójával. A Borland C++ 3.0 rendszer az AT&T C++ 2.1-es változatával, míg a Borland C++ 3.1 az AT&T C++ 3.0-ás verziójával kompatibilis.

Mielőtt elkezdenénk a különbségek részletes tárgyalását, le kell szögeznünk, hogy a C++ nyelv a C nyelv szintakszisára épülő önálló programozási nyelv. Alapvető eltérés a két nyelv között, hogy amíg a C nem típusos nyelv, addig a C++ erősen típusos objektum-orientált nyelv.

A különbségek ismertetését két lépésben végezzük. Először áttekintjük azokat az eszközöket, amelyek a C nyelv természetes kiegészítését jelentik. Ezen lehetőségek felhasználásával hatékonyan készíthetünk nem objektum-orientált programokat. A második részben az objektum-orientált programozást támogató C++ nyelvvel ismerkedünk meg. A Windows alkalmazások írásakor, az ObjectWindows könyvtárat használva, a két részben elmondott ismereteket együttesen kell felhasználnunk.

A Borland C++ szintén lehetőséget kínál a fenti részek szétválasztására a ***CPP Extensions Only*** és a ***C++ Always*** fordítási opciók beállításával. Az első esetben .C kiterjesztésű file-ok esetén csak bizonyos C kiegészítések használatát engedélyezi a fordító, míg a második opció választásakor a teljes C++ nyelvdefiníció használható. A .CPP kiterjesztésű file-ok esetén mindkét esetben C++ programként fordít a fordító.

Sokan bírálják a C++ fordítókat, hogy nagyon lassúak a C nyelvi fordítókhoz képest. Már a legelején érdemes megérteni, hogy a C++ fordító összehasonlíthatatlanul több feladatot végez el a fordítás során, mint a hagyományos C fordítóprogram, biztonságossá téve ezáltal a C++ program fejlesztését.

F3.2 A C++ mint egy jobb C nyelv

F3.2.1 A C++ első látásra

Tekintsük az alábbi egyszerű C nyelven megírt programot, amely bekér egy szöveget és két számot, majd kiírja a szöveget és a számok szorzatát.

```
#include <stdio.h>
main()
{
 char nev[ 20] ;
 int  a;
 double b;

 printf("Kérem a szöveget: ");
 scanf("%s",nev);
 printf("A=");
 scanf("%d",&a);
 printf("B=");
 scanf("%lf",&b);
 printf("%s : A*B=%lf\n",nev,a*b);
}
```

A példában a I/O műveletek elvégzéséhez a szabványos C-könyvtárban található *scanf()* és *printf()* függvényeket használtuk. Ezen függvények nem tekinthetők a C nyelv részének, hiszen csak könyvtári függvények. A C++ nyelv a szabványos I/O műveletek kezelésére szintén tartalmaz kiegészítést, a a **cin** és a **cout** *stream*-ek definiálásával, amelyek szintén nem képezik részét a C++ nyelv definíciójának.

Ezen osztályok felhasználásával a fenti program igazi C++ alakja:

```
#include <iostream.h>
void main()
{
 char nev[ 20] ;
 int  a;
 double b;

 cout << "Kérem a szöveget: ";
 cin  >> nev;
 cout << "A=";
 cin  >> a;
 cout << "B=";
 cin  >> b;
 cout << nev <<" : A*B=" << a*b;
}
```

Szembeötlő eltérés a C programhoz képest, hogy az I/O műveletek használata egyszerűbbé vált. Nem kell figyelni a megfelelő formátum megadására, illetve a helyes paraméterezésre, mindezt elvégzi helyettünk a fordító program.

A szabványos input elvégzésére a **cin** *stream*-et, míg a szabványos outputként a ***cout*** *stream*-et használhatjuk. Létezik még egy szabványos hiba *stream* is, a **cerr**. Mindhárom *stream* definícióját a IOSTREAM.H *include* file tartalmazza.

A *stream*-osztályok lehetőségeit részletesen a következő fejezet után tárgyaljuk. Az itt bemutatott szabványos I/O műveleteket a példaprogramokban kívánjuk felhasználni.

F3.2.2 Megjegyzések használata

A C++ nyelvben a megjegyzések elhelyezésére a programban a /* ... */ jeleken kívül a // (két perjel) is használható. A // jel használata esetén a megjegyzést nem kell lezárni, hatása a sor végéig terjed.

```
/* Az alábbi részben megadjuk
   a változók definícióit     */
int i=0;      /* segédváltozó */

// Az alábbi részben megadjuk
// a változók definícióit
int i=0;      // segédváltozó
```

A C++-ban mindkét programrészlet helyes. A // jel támogatja a jól dokumentált programok írását.

F3.2.3 A C++ nyelv kulcsszavai

Az alábbiakban összefoglaltuk azokat a foglalt kulcsszavakat, amelyekkel a C++ nyelv kibővült.

class	**delete**	**friend**	**inline**	**new**	**operator**
private	**protected**	**public**	**template**	**this**	**virtual**

F3.2.4 Típusok, deklarációk

F3.2.4.1 A char konstans típusa

C nyelvben a **char** konstans *'K'* típusa **int**, míg a C++-ban **char**. Ebből következik, hogy a *sizeof('K')* kifejezés értéke C-ben 2, míg C++-ban 1.

F3.2.4.2 Különbségek az enum típus használatában

Az

```
enum szin { fekete, kek, zold} ;
```

deklarációt használva, az alábbi programrészlet C++-ban figyelmeztetéshez vezet.

```
enum szin col;
int  kod;

col=zold;      // rendben
kod=col;       // rendben, a kod értéke 2 lesz
col=26;        // hibás!
```

További különbség van a felhasználói adattípuson (**struct**) belüli **enum** deklaráció értelmezésében. A

```
struct fest {
enum szin { fekete, kek, zold} ecset;
} ;
int kek;
```

deklaráció hatására a C fordító hibajelzést ad, hogy a *kek* változó többször definiált. A C++ azonban, az **enum** deklarációt a struktúrán belül lokálisan kezeli.

További eltérés, hogy az **enum** deklarációban megadott név típusnévként is használható az **enum** kulcsszó megadása nélkül:

```
enum Boolean { false, true} ;

Boolean x=false;
```

F3.2.4.3 Konstansok a C++-ban

A C nyelvben létezik ugyan a **const** definíció, de valójában ez is egy változó, amelyhez nem enged közvetlen hozzáférést a fordító (indirektet azonban igen).

```
const int ci=1970;
int * pi;

ci=1993;  // Hibás C- és C++-ban
pi=&ci;   // csak C++-ban hibás
*pi=1993;
```

A C++ fordító sokkal szigorúbban ellenőrzi a **const** típusú konstansok felhasználását:

```
const double pi=3.14.159265; // szabályos konstansdefiníció

const double ac=1.26;
double * pd = &ac;           // Hiba!

const double *pdc;           // double konstansra mutató pointer
const double dc=1.26;
double   d;
pdc  = &dc;                  // a pdc pointer a dc-re mutat
pdc  = &d;                   // a pdc pointert a d-re állítjuk
*pdc = 6.28;                 // Hiba! A pointer segítségével a d
                             // változó sem változtatható meg.
int ev;
int * const aktev=&ev;      // az aktév pointer értéke nem
                           // változtatható meg, de a *aktev
                           // módosítható:
*aktev=1993;

const int ev=1970;
const int ae=1993;
const int * const aktev=&ev;// int típusú konstansra mutató
                           // konstans pointer;

aktev = &ae;                //Hiba!
*aktev= 1993;               //Hiba!
```

Ugyancsak a **const** típusú konstansok használata mellett szól az a lehetőség is, hogy tömbök definíciójában is felhasználhatók:

```
const maxnum=100;
int num[ maxnum] ;
```


F3.2.4.4 A hivatkozási (referencia) típus és használata

A hivatkozási típus felhasználásával már létező változókra hivatkozhatunk, alternatív nevet definiálva. A definíció általános formája:

```
típus & azonosító = objektum;
```

Látható, hogy referencia definiálásakor kötelező kezdőértéket adnunk. Nézzünk néhány példát a hivatkozási típus használatára:

```
double   n;
double & m=n;      // m az n változó alternatív (alias) neve

int a[ 10] ;
int &last=a[ 9] ;   // a last hivatkozás az utolsó
                   // tömbelemre

long & la=a[ 0] ;   // az la NEM hivatkozik az első két
                   // tömbelemre!

char & nl ='\n';
```

A hivatkozás egyszerűen az új nevek felhasználásával történik:

```
m=26;
a[ 9] =m+23;
la=19701993;
```

Az utolsó két példában, amikor hivatkozás típusa különbözik az hivatkozott objektum típusától, illetve ha konstanst használunk inicializálásra, a fordító ideiglenes változókat (*T1*, *T2*) hoz létre, amelyekre hivatkoznak az *la* illetve az *nl* változók. A

```
long & la=a[ 0] ;
```

helyett a

```
long T1=long(a[ 0] );
long & la = T1;
```

míg a

```
char & nl ='\n';
```

helyett a

```
char T2='\n';
char &nl=T2;
```

utasításokat fordítja a fordító.

Fontos megértenünk, hogy a referencia nem azonos a mutató típussal, amely természetesen a C-ben megszokott módon továbbra is használható.

```
int a;          // egy egész típusú változó
int &ra=a;      // referencia az a változóra
int *pa=&a;     // mutató az a változóra;

ra=4;           // értékadás az a változónak az alias nevének
                // felhasználásával
*pa=26;         // értékadás a mutatójának felhasználásával.
```

A *pa* értéke (ezáltal a mutatott objektum) bármikor megváltoztatható, míg az *ra* az *a* változóhoz kötött.

Referencia típus szintén definiálható a **typedef** utasítás felhasználásával. Az előző példa *ra* hivatkozását az alábbiak szerint is létrehozhatjuk:

```
typedef int & tri;  // tri hivatkozás egészre típus
tri ra=a;           // nem szabad megfeledkeznünk az
                    // inicializálásról
```

A referencia típus igazi lehetőségét a függvényeknek történő paraméterátadás új módja, a hivatkozás szerinti paraméterátadás jelenti. (A C nyelv csak az érték szerinti paraméterátadást ismeri, ezért használunk mutatókat, ha valamely változót függvényen belül kívánunk megváltoztatni.)

Klasszikus feladat két egész típusú változó tartalmának felcserélése, amelyet mutatók segítségével a *pswap()*, míg referencia felhasználásával a *rswap()* függvénnyel oldottunk meg.

```
void pswap(int * pa, int *pb)
{
   int temp;
   temp=*pa;
   *pa=*pb;
   *pb=temp;
}
void rswap(int & a, int & b)
{
   int temp;
   temp=a;
   a=b;
   b=temp;
}
```


A függvények hívását az alábbi **main()** tartalmazza:

```
void main()
{
 int x=4, y=26;
 pswap(&x,&y);
 rswap(x,y);
}
```

Referenciát készíthetünk tetszőleges típusú objektumhoz. A helyes típusmegadás érdekében érdemes a **typedef** definíciót használnunk.

```
// referencia pointerhez
typedef int *tp;
int * pa;
tp & px=pa;              // referencia a pa mutatóra
int * & py=pa;           // referencia a pa mutatóra;

int a;
px=&a;                   // értékadás a pa mutatónak
*py=126;                 // értékadás az a változónak

// referencia függvényhez
typedef void tf(int &, int&);
tf & fx=rswap;           // referencia az rswap() függvényhez
fx(x,y);                 // az alternatív név felhasználása
```

Hivatkozási típus felhasználásával készíthetünk olyan függvényt, amely referenciát ad vissza:

```
int & refv(int & x)
{
 return x;
}
// a függvény használata
void main()
{
  int a=126;
  refv(a)=94;  // az a értéke 94 lesz
}
```

F3.2.4.5 A változók deklarációja

A C nyelvben a változók deklarációját a blokk elején az utasítások előtt kell elhelyeznünk. A C++ megengedi, hogy a változók deklarációját bárhova helyezzük a program kódján belül. Egyetlen feltétel, hogy a változót mindenképpen deklarálnunk kell a felhasználása előtt. Élve ezzel a

lehetőséggel a változó deklarációja és a felhasználása közel helyezhető, elkerülve ezzel bizonyos programozási hibákat.

```
// Változók deklarációja az első felhasználás közelében
#include <iostream.h>
void main()
{
 cout << "Kerek egy számot: ";
 int n;
 cin  >> n;
 for (int i=1; i<=n; i++)
   cout << "\n" <<i;
}
```

A példában a **for** ciklusba helyeztük a ciklusváltozó deklarációját. Egy következő for ciklus esetén a *int i=1;* kezdőértékadás már nem használható, hiszen az *i* változót már deklaráltuk. Ha a ciklusváltozót lokálissá kívánjuk tenni, a ciklust egy blokkba kell helyeznünk:

```
{   for (int i=1; i<=n; i++)
      cout << "\n" <<i;
}
```

Az alábbi két esetben azonban, a deklaráció nem vihető be a kifejezésbe:

```
if (int s==0)       //Hibás!
   ;

while (int w==12)   //Hibás!
  ;
```

F3.2.4.6 A felhasználó által definiált típusok

A C nyelvben a **struct** és az **union** kulcsszavakkal definiálhatunk felhasználói adattípust. A C++-ban ez a lehetőség kibővült a **class** definíció bevezetésével.

Tekintsük az alábbi struktúra definíciót.

```
struct cmplx
   {
     double re,im;
   };

struct cmplx a,b ;     // a szokásos C változó definíció
cmplx a;               // C++ változó definíció
```


A C++ nyelvben a **struct**, **union** és **class** kulcsszavak után álló név típusnévként használható a kulcsszó megadása nélkül.

A **class** deklarációt osztályok létrehozásához használjuk. A C++-ban a **struct** és a **union** szintén osztályt definiál. Miért volt szükség akkor új kulcsszó bevezetésére? A magyarázat az osztály adattagjainak (és tagfüggvényeinek) elérhetőségében keresendő. A **public**, a **private** és **protected** deklarációk megadása nélkül (alapértelmezés szerint), a **class** típusú osztály adattagja kívülről nem érhetők el (**private**), míg a **struct** típusú osztály adattagjai elérhetők (**public**). Ez az elérhetőség természetesen szabályozható a **public** és **private** mezők deklarálásával. **Union** estén a **public** elérhetőség nem állítható át. A

```
struct cmplx
   {
     double re,im;
   };
```

definícióval ekvivalens a

```
class cmplx
    {
    public:
      double re,im;
    };
```

definíció.

A felhasználói (absztrakt) adattípusokkal F3.3. fejezetben részletesen foglalkozunk, hiszen ezen alapulnak a C++ nyelv objektum-orientált lehetőségei.

F3.2.4.7 Névtelen union-ok használata

A névtelen **union**-okat általában struktúrák adattagjainak helytakarékos tárolásához, illetve speciális konverziós lépések elvégzéséhez használjuk. A C nyelv nem teszi lehetővé a névtelen **union**-ok használatát, mint ahogy az a következő példaprogramban is látható.

```
#include <stdio.h>

struct msg {
  union {
      long mdword;
      struct {unsigned Lo,Hi;} dws;
  } mu;
} st;

main()
{
   st.mu.mdword=0x1261970L;
   printf("%04x  %04x\n",st.mu.dws.Lo, st.mu.dws.Hi);
}
```

A példában az *st* struktúrán belüli **union**-t *mu* névvel kellett ellátnunk, ezért az *mdword* és a *dws* adattagokra mint az *mu* **union** mezeire hivatkoztunk.

A C++ nyelvben névtelen **union**-t használva, az *mdword* és a *dws* adattagok az *st* struktúra tagjaként használhatók:

```
#include <stdio.h>

struct msg {
  union {
      long mdword;
      struct {unsigned Lo,Hi;} dws;
  };
} st;

void main()
{
   st.mdword=0x1261970L;
   printf("%04x  %04x\n",st.dws.Lo, st.dws.Hi);
}
```

F3.2.4.8 Típuskonverzió a C++-ban

Az implicit konverziók a C nyelvben megszokott módon mennek végbe, kivéve az alábbiakat:

- Aritmetikai kifejezésben a **float** típus nem konvertálódik automatikusan double típussá. Az alábbi kifejezés típusa **float**:

```
int i=4;
float f=3.1415,g;
g=i*f-3; // a jobb oldal float típusú!
```


- Nincs implicit pointer konverzió! Az alábbi programrészlet fordításakor hibát jelez a fordító:

```
int *p;
char *q;
p=q;      // hibás sor! int * <-> char *
```

Ha mégis a fenti konverzióra van szükségünk, explicit típuskonverzió segítségével a program lefordítható:

```
p=(int *)q;
```

Tetszőleges típusú mutató automatikusan **void** * típusúvá a nulla (0) érték tetszőleges típusú pointerré konvertálódik.

```
int  *ip=0;     // helyes utasítások
void *vp=ip;
ip=(int *)vp;  // a visszaalakításhoz azonban már explicit
               // konverziót kell használni!
```

- Az explicit típuskonverzió szerepe megnőtt a C++-ban. A C nyelvben az explicit konverzió alakja (típus)kifejezés:

```
(int) 3.14*a;
```

Ez természetesen használható a C++ nyelvben is, azonban helyette a *típus (kifejezés)* alak használata javasolt:

```
int(3.14*a);
```

Természetesen a bonyolultabb típuskifejezések használata esetén, mint például az *(int *)p*, nem használható az *int * (p)* alak. Ekkor a **typedef** segítségével először új névvel kell ellátnunk a típust:

```
typedef int * intptr;
```

Ezek után már minden további nélkül alkalmazható az új formátumú típuskonverzió:

```
intptr(p);
```

Az újfajta típuskonverzió használatát az is indokolja, hogy a felhasználói típusokhoz saját konverziót is definiálhatunk (F3.3 fejezet).

F3.2.5 Kifejezések és operátorok

A C++ nyelv több operátorral bővíti a C nyelvet, melyek többségének igazi felhasználási területe az osztályokkal áll kapcsolatban:

::	érvényességi kör operátor
new	dinamikus memóriafoglalás operátora
delete	dinamikus memóriafelszabadítás operátora
.*	osztálytagra történő indirekt hivatkozás operátora
->*	mutatóval megadott osztályobjektum tagjára való indirekt hivatkozás operátora

Az utolsó két operátort az osztályokat tárgyaló fejezetben ismertetjük.

F3.2.5.1 Az érvényességi kör operátor ::

Az érvényességi kör operátor kettős szerepet tölt be a C++ nyelvben. Az első értelmezésében a :: operátor segítségével a program tetszőleges blokkjából hivatkozhatunk a globális (file) érvényességi körrel rendelkező nevekre.

```
int i=126;

void main()
{
   double i=3.14;

   {
    long i=5,a;
    a=i*::i;
    // az a változó értéke 5*126=630
   ::i=94;
   }
  // a ::i változó értéke 94
}
```

Az érvényességi kör operátort használjuk akkor is, amikor valamely osztály adattagjaira, illetve tagfüggvényeire hivatkozunk (F3.3 fejezet).

F3.2.5.2 A new és a delete operátorok használata

A szabad memória dinamikus használata alapvető részét képezi minden C nyelven megírt programnak. A C programokban könyvtári függvényekkel

végezhetjük el a szükséges memóriafoglalási *(malloc(),...)* illetve felszabadítási *(free())* műveleteket. C++ nyelvben a **new** és **delete** operátorok nyelvdefiníció szintjén helyettesítik a fenti C könyvtári függvényeket.

A **new** operátor az operandusában megadott típusnak megfelelő méretű területet foglal a szabad memóriában, és a területre mutató pointert ad eredményül:

```
int * ip;
ip=new int;
```

A ***delete*** operátor a **new** operátor által lefoglalt területet felszabadítja:

```
delete ip;
```

Nézzünk néhány példát a **new** használatára.

1. Helyfoglalás 10 elemű egész tömb számára:

```
int *ap;
ap=new int [ 10] ;
```

2. Helyfoglalás 10 elemű egészre mutató pointer tömb számára:

```
int **pa;
pa=new int * [ 10] ;
```

3. A memóriafoglalás ellenőrzés beépítésével:

```
#include <iostream.h>

int main()
{
   long * data;
   long size;

   cout << "\nKérem a tömb méretét: ";
   cin >> size;
   // Memóriafoglalás
   data = new long [ size] ;
   // A foglalas sikerességének ellenőrzése
   if ( !data )
   {
      cerr << "\nNincs elég memória !\n" << endl;
      return -1;
   }
```

```
    for (long i=0; i<size; i++)
       data[ i] =i+1;

    for (i=0; i<size; i++)
      cout << data[ i] << "\t";

    cout << "\n\n";

    // A lefoglalt memória felszabadítása
    delete data;

    return 0;
}
```

A *malloc()* és a *free()* függvények kompatibilitási megfontolások miatt továbbra is elérhetők a C++ nyelvből, de helyettük ajánlott a **new** és a **delete** operátorokat alkalmazni. Ha a **new** és a **delete** operátorokkal foglalunk helyet valamely osztályobjektum számára, akkor a rendszer automatikusan meghívja az osztály konstruktorát a **new**, illetve a destruktorát a **delete** művelet végrehajtásakor.

A teljesség kedvéért közöljük az új C++ operátorokat is tartalmazó operátor precedencia táblázatot. A táblázatban feltüntettük az azonos precedenciával rendelkező operátorok kiértékelési irányát is (asszociativitás).

1. Legmagasabb	*asszociativitás: balról->jobbra*	
	()	függvényhívás
	[]	tömbindexelés
	->	C++ indirekt komponens kiválasztó op.
	::	C++ érvényességi kör op.
	.	C++ direkt komponens kiválasztó op.
2. Egyoperandusú	*asszociativitás: jobbról->balra*	
	!	logikai tagadás (NOT)
	~	bitenkénti negálás
	+	+ előjel
	-	- előjel
	++	léptetés előre
	--	léptetés vissza
	&	a címe op.
	*	indirektség op.
	sizeof	(az operandus byte-ben kifejezett méretét adja meg)
	new	(C++ tárterület dinamikus allokálása)
	delete	(C++ tárterület dinamikus felszabadítása)

3. Osztály tagjainak elérése	*asszociativitás: balról->jobbra*	
	.*	(C++ osztálytagra történő indirekt hivatkozás operátora)
	->*	(C++ mutatóval megadott osztályobjektum tagjára való indirekt hivatkozás operátora)
4. Multiplikatív	*asszociativitás: balról->jobbra*	
	*	szorzás
	/	osztás
	%	maradék
5. Additív	*asszociativitás: balról->jobbra*	
	+	összeadás
	-	kivonás
6. Biteltolás	*asszociativitás: balról->jobbra*	
	<<	eltolás balra
	>>	eltolás jobbra
7. Összehasonlító	*asszociativitás: balról->jobbra*	
	<	kisebb
	<=	kisebb vagy egyenlő
	>	nagyobb
	>=	nagyobb vagy egyenlő
8. Egyelőség	*asszociativitás: balról->jobbra*	
	==	egyenlő
	!=	nem egyenlő
9. - 13.	*asszociativitás: balról->jobbra*	
9.	&	bitenkénti AND
10.	^	bitenkénti XOR
11.	|	bitenkénti OR
12.	&&	logikai AND
13.	||	logikai OR
14. Feltételes	*asszociativitás: jobbról->balra*	
	?:	(a ? x : y jelentése: "ha a akkor x, különben y")
15. Értékadás	*asszociativitás: jobbról->balra*	
	=	egyszerű értékadás
	*=	szorzat megfeleltetése
	/=	hányados megfeleltetése
	%=	maradék megfeleltetése
	+=	összeg megfeleltetése
	-=	különbség megfeleltetése
	&=	bitenkénti AND megfeleltetése
	^=	bitenkénti XOR megfeleltetése

	|=	bitenkénti OR megfeleltetése
	<<=	balra eltolt megfeleltetése
	>>=	jobbra eltolt megfeleltetése
16. Vessző		*asszociativitás: balról->jobbra*
	,	kiértékelés

A felhasználói típushoz kapcsolódóan a C++ lehetőséget kínál az operátorok többségének átdefiniálására (*operator overloading*). Kivételt képeznek ez alól a

. .* :: ?:

operátorok. Az átdefiniálás során az egyes operátorok új értelmezést kaphatnak, azonban a fenti táblázat szerinti precedencia és asszociativitás nem változtatható meg.

F3.2.6 Függvények

F3.2.6.1 A függvények prototípusa

A C nyelvben deklaráltunk egy függvényt, anélkül, hogy azt definiáltuk volna. Ezt a függvénydeklarációt, amely tartalmazza a függvény nevét, típusát és paramétereinek típusát, a függvény prototípusának nevezzük.

A függvények prototípusának használata C++ nyelvben kötelező. Az alábbi programot csak akkor tudjuk lefordítani, ha a *main()* függvény elé beszúrjuk a

```
void display(char *);
```

sort, vagyis a *display()* függvény deklarációját (prototípusát).

```
#include <iostream.h>
void main()
{
 display("Hello\n");
}
void display(char *s);
{
 cout <<s;
}
```


F3.2.6.2 Különbségek a függvények prototípusának értelmezésében

Az alábbi programrészlet C++ fordítóval nem fordítható le, mivel eltér a C és a C++ nyelvben az *f()* értelmezése.

```
extern int f();
void main()
{
  int k;

  k=f(26);
}
```

Összefoglaltuk néhány speciális paraméter nélküli függvénydeklaráció értelmezését.

deklaráció	*C értelmezés*	*C++ értelmezés*
`f()`	`f(...)`	`f(void)`
`f(...)`	`f(...)`	`f(...)`
`f(void)`	`f(void)`	`f(void)`

F3.2.6.3 Alapértelmezés szerinti (default) függvényargumentumok

A C++ függvények prototípusában bizonyos paraméterekhez ún. alapértelmezés szerinti értéket rendelhetünk. A fordító ezeket az értékeket használja fel a függvény hívásakor, ha az adott argumentum nem szerepel a hívási listában:

```
double defargfv(int a, double b=3.14, char c='K');
```

A példából is látható, hogy az alapértelmezés szerinti értékkel ellátott paraméterek jobbról-balra haladva folytonosan helyezkednek el. A fenti függvény lehetséges hívásait felsorolva nézzük meg a paraméterek tényleges értékeit:

Hívás	Paraméterek:	a	b	c
`defargfv(10);`		10	3.14	'K'
`defargfv(10,2.5);`		10	2.5	'K'
`defargfv(10,2.5,'X');`		10	2.5	'X'

Nem megengedett hívási forma például a *defargfv(10,,'X');*. Ha valamely default argumentumot elhagyjuk, akkor az azt követő alapértelmezés szerinti értékkel ellátott argumentumokat is el kell hagynunk.

A következő példában használt *terulet()* függvény háromszög területét határozza meg az oldalak ismeretében. (Derékszögű háromszög esetén a két befogó felhasználásával a terület egyszerűbben meghatározható.)

```
#include <math.h>
#include <iostream.h>

double terulet(double a, double b, double c=0);

void main()
{
 // Általános háromszög területe:
 cout <<"\nÁltalános : " << terulet(3,4,5);

 // Derékszögű háromszög területe:
 cout <<"\nDerékszögű: " << terulet(3,4);

}

double terulet(double a, double b, double c)
{
 if (c)
  {
    double s=(a+b+c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
  }
  else
    return(a*b/2);

}
```

Az alapértelmezés szerinti argumentumokkal ellátott prototípusok rugalmassá teszik a függvények használatát. Például, ha valamely függvényt sokszor hívunk egyazon argumentumlistával, érdemes a használt argumentumokat alapértelmezés szerintivé tenni, és a hívást argumentumok megadása nélkül elvégezni.

F3.2.6.4 Inline függvények

A C előfordító (preprocesszor) **#define** direktívájának használata során olyan hibákat vihetünk be a programunkba, amelyek a forráskód áttanulmányozásával nem derülnek ki (kifejezés megadása makróban, vagy a léptető

operátorok használata a makró paraméterében). A C++-ban javasolt a **#define** használatának korlátozása, hiszen az esetek többségében a **const** és **inline** definíciók kiváltják azt.

Az **inline** ("soron belüli") függvényekkel a **#define** makrókat helyettesíthetjük. Az **inline** függvény esetén a függvény törzsét képező kód helyettesítődik be a függvényhívás helyére ("kódmakró"). Például a *MAX* makró helyett egész értékek esetén a max() **inline** függvény használata javasolt.

```
#define MAX(x,y) (x)>(y)?(x):(y)

inline int max(int x, int y)
{
 return (x>y?x:y);
}
```

Nézzük a hivatkozást a makróra és az **inline** függvényre:

```
void main()
{
 int a;
 a=MAX(4,26); // az előfordító a=(4)>(26)?(4):(26); utasítássá
              // alakítja át.

 a=max(4,26); // a fordító a függvény törzsét képező kódot
              // helyettesíti
              // be az utasításba: a=x>y?x:y;
}
```

Tudnunk kell azonban, hogy az **inline** definíció csak javaslat a fordító számára, amelyet bizonyos feltételek esetén (de nem mindig!) figyelembe vesz. Az **inline** megoldás előnye, hogy a függvényhíváskor az argumentumok feldolgozása teljes körű típusellenőrzés mellett megy végbe. Talán ez egyben a hátránya is, hiszen a *MAX* makró tetszőleges aritmetikai típus esetén használható, míg a max() függvény csak **int** típusú argumentummal. Ezen probléma kiküszöbölésében a C++ egy nagyon fontos mechanizmusa, a függvénynevek átdefiniálása (*overloading*) segít.

F3.2.6.5 Függvénynevek átdefiniálása (overloading)

A C++-ban több függvényt is definiálhatunk ugyanazzal a névvel, ha a definiált függvények paraméterlistája ("kézjegye") eltér egymástól. Az így definiált függvények közül az éppen szükségeset a fordító választja ki a hívási argumetumok száma és típusa alapján.

Az alábbi példában a *sumall()* függvénynek két átdefiniált formája létezik, az **int** és a **double** típusú tömbök elemösszegének meghatározására.

```
#include <iostream.h>

int sumall(int a[], int n)
{
 int sum;
 for (int i=0; i<n; i++)
  sum+=a[i];
 return sum;
}

double sumall(double a[], int n)
{
 double sum;
 for (int i=0; i<n; i++)
  sum+=a[i];
 return sum;
}

void main()
{
  int        ai[]={1,1,2,3,5,8,13};
  double     ad[]={1.2,2.3,3.4,4.5,5.6};
  const int  ni=sizeof(ai) / sizeof(ai[0]);
  const int  nd=sizeof(ad) / sizeof(ad[0]);

  cout << "\nAz   int tömb elemösszege: "<<sumall(ai,ni);

  cout << "\n\ double tömb elemösszege: "<<sumall(ad,nd);

}
```

A fordító az első esetben *sumall(int *, const int)*, míg a másodikban *sumall(double *, const int)* szignatúrát talál. Ezért például **unsigned** és **float** tömbök esetén fordítási hibát kapunk, hiszen C++-ban a mutatók automatikus konverziója erősen korlátozott.

A megfelelő függvény kiválasztása több lépésben megy végbe:

1. Keresés a teljes (egzakt) típusegyezőség feltételének felhasználásával. A teljes típusegyezőség esetén a **float** nem egyezik meg a **double**, míg a **char** nem egyezik meg az **int** típussal.

2. Keresés az ún. triviális típuskonverziók végrehajtásával. A C++ nyelvben az alábbi konverziókat tekintjük triviálisnak:

Típusról	*Típusra*
típus	típus&
típus&	típus
típus[]	típus*
típus	const típus
típus	volatile típus
fv(argumetumok)	(*fv)(argumentumok)

   Az utolsó konverzió a függvény neve és függvényre mutató pointer közötti automatikus konverziót jelenti.

3. Keresés az egész típusok közötti konverzió, illetve a **float** -> **double** konverzió felhasználásával.

4. Keresés a szabványos típuskonverziók alkalmazásával. Ezek a konverziók alapvetően a szokásos aritmetikai (**int** -> **double**, **unsigned** -> **signed**), és a mutató konverziókat jelentik. A pointer konverziókat az alábbiakban foglaltuk össze:

Típusról	*Típusra*
tetszőleges mutatótípus	void *
leszármaztatott osztály mutatója	az alaposztályra mutató pointer,
0 konstans	nulla pointer.

   A referencia típusok konverziója a mutató típusok konverziójának megfelelően megy végbe.

5. A típusegyezőség keresése ideiglenes objektum létrehozásával.

6. Keresés a felhasználó által definiált konverziók alkalmazásával.

A függvényekhez hasonló módon használható a műveletek átdefiniálásának mechanizmusa (*operator overloading*). Az operátorokat azonban csak felhasználó által definiált típusokkal lehet átdefiniálni (**struct**, **class**), ezért ezzel a lehetőséggel a következő részben foglalkozunk részletesen.

F3.2.6.6 Típusmegőrző szerkesztés (type-safe linking)

A C++ 2.0-ás definíciója rendelkezik a címben említett koncepcióval, mely segíti a hibás argumentummal történő függvényhívások kiszűrését a szerkesztés folyamán. Az alábbi példában két modulból épül fel a C programunk:

```
/* File: MYMATH.C */
#include <math.h>

double mysqrt(double d)
{
/* a gyökvonás megvalósítása */
return sqrt(d);
};

/* File: MYPROG.C */

#include <stdio.h>

extern double mysqrt(int); /* hibás deklaráció! */
void main()
{
double dv;
int    iv=64;

dv=mysqrt(iv);
printf("%lf\n",dv);
}
```

C fordítóval a két modult lefordítva, a szerkesztés minden további nélkül végbemegy, csak éppen a program hibásan működik. Ezen nem kell csodálkozni, hiszen mindkét modulban a *_mysqrt* név kerül bele a tárgymodulba, azonban hibás paraméterezéssel.

Ha a fordítást C++ fordítóval C++ módban végezzük, teljesen megváltozik a helyzet. A C++ fordító a függvények tárgymodulbeli nevének kialakításakor a függvény paraméterezéséről is tárol információt. A Borland C++ rendszerben

a MYMATH.C file-ban található *mysqrt()* függvény neve *@mymath$qd* lesz, míg a MYPROG.C file esetén *@mymath$qi*. (A *$q* utáni *d* betű a **double** típusú, míg az *i* betű az **int** típusú paramétert jelöli.) Ezek után teljesen természetes, hogy a szerkesztő (*linker*) hibajelzéssel fog leállni, hisz a két függvénynév különbözik egymástól.

A C++ fordítók ilyen névképzése teszi lehetővé a függvénynevek átdefiniálását, hisz ekkor a függvény tényleges nevének kialakítása függ a függvény paraméterezésétől. Megjegyezzük, hogy valamely osztály tagfüggvényeinek nevei az osztály nevét is tartalmazzák.

A függvénynevek ilyen formán történő képzése természetesen megnehezíti más nyelven (például C) írt modulok, könyvtárak C++ nyelvből való hasznosítását. Ezen probléma leküzdésére a C++ nyelv tartalmazza az **extern "C"** típusmódosítót. Ha például C nyelven megírt *sin()* függvényt szeretnénk használni a C++ programból, szükséges az

```
extern "C" double sin(double a);
```

deklaráció megadása. Ennek hatására a C++ fordító a szokásos C névképzést használja a *sin()* függvényre. Ebből persze az is következik, hogy a nem C++ függvényeket nem lehet átdefiniálni!

Ha több C függvényt kívánunk meghívni, akkor az extern "C" deklaráció csoportos formáját használhatjuk:

```
extern "C" {
double sin(double a);
double cos(double a);
}
```

Ha a C könyvtár függvényeit külön *include* file tartalmazza, akkor a következő megoldás áll rendelkezésünkre:

```
extern "C" {
#include <stdio.h>
}
```

F3.3 A C++ mint objektum-orientált nyelv

Az objektum-orientált programozás (OOP) a gondolkozást, cselekvést közelítő programozási mód. Az objektum-orientált programozási nyelv sokkal strukturáltabb, modulárisabb és absztraktabb, mint egy hagyományos programozási nyelv.

Ellentétben a hagyományos programozási nyelvekkel, mint például a C, nem a műveletek (funkciók) megalkotása áll a programozás központjában, hanem az egymással kapcsolatban álló programegységek hierarchiájának megtervezése.

Míg a hagyományos programozási nyelvek használata során az adatok csak másodlagos szerepet töltenek be a rajtuk elvégzendő műveletekkel (függvények) szemben, addig az OOP nyelvben az adatokat és adatokon elvégzendő műveleteket egyenrangúan, zárt egységben kezeljük. Ezeket az egységeket objektumoknak hívjuk. Az adatok és az adatokat kezelő függvények (*metódus*) egységbezárása (*encapsulation*) nagyon fontos sajátossága minden objektum-orientált nyelvnek. A C++-ban az objektumoknak megfelelő tárolási egység típusát osztálynak (***class***) nevezzük. Az objektum tehát valamely osztálytípussal definiált változó, amelyet más szóhasználattal az osztály példányának nevezünk (*instance*).

Az így kialakított osztályok további, a nagy méretű programok kialakításához elengedhetetlen lehetőséggel, az adatrejtés (*data hiding*) képességével rendelkeznek. Ez azt jelenti, hogy az osztály tagjainak elérhetősége szabályozható a **private** és a **public** kulcsszavak felhasználásával.

```
// a vector osztály deklarációja
class vector {
  private:           // private elérésű adattagok
     int x,y;
  public:            // public elérésű tagfüggvények (metódusok)
     void init(int a,int b);
     int getx();
     int gety();
};

// a vector osztály definíciójához meg kell adnunk az összes
// tagfüggvény definícióját!
```



```
void main()
{
  //a vektor típusú objektumpéldány létrehozása
 vector v1;

 // hivatkozás az objektum mezőire
 v1.init(3,4);  // a public elérésű init() tagfüggvény hívása - OK.
 v1.x=4;        // HIBA! Az x és y adatmezők kívülről nem érhetők el.

}
```

A C++ nyelvben definiált osztálytípus a nyelv szerves részévé tehető, az ún. operátor *overloading* (operátor átdefiniálás) mechanizmusának felhasználásával. Ennek segítségével az általunk létrehozott típushoz definiálhatunk operátorokat, konverziókat sőt akár specifikus I/O műveleteket is. Ugyancsak ezt a célt szolgálják az objektum automatikus inicializálását elvégző konstruktorok, illetve az objektum lebontásában fontos szerepet játszó destruktor használatának lehetősége.
Ezen eszközök felhasználásával az általunk létrehozott absztrakt adattípus (*ADT* - osztály) a C++ nyelv természetes bővítéseként használható.

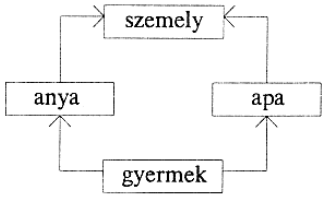
Az objektum-orientált nyelvek másik fontos sajátossága az öröklődés (*inheritance*) lehetősége. Az öröklődés azt jelenti, hogy már meglévő osztály(ok)ból kiindulva újabb osztályt építhetünk fel, amely örökli a felhasznált osztály(ok) adattagjait és tagfüggvényeit. A C++-ban azt az osztályt amelyből az új osztályt származtatjuk ős vagy alap (*base*) osztálynak, míg az új osztályt leszármaztatott (*derived*) osztálynak nevezzük. A C++ 2.0-ás változatától kezdődően használható a többszörös öröklődés (*multiply inheritance*) mechanizmusa, amikor az új osztály származtatása során több alaposztályból indulunk ki.
Az öröklődés során tovább szűkíthető, illetve megőrizhető az alaposztály tagjainak elérhetősége a **private** és a **public** öröklődés felhasználásával. Az öröklődés lehetővé teszi, hogy az egyedi objektumok helyett a problémák leírására objektumok egymásra épült hierarchiáját használjuk.

```
// alaposztály
class szemely {
};
// egyszeres öröklődés: a személy osztályból származtatott osztály
class anya : public szemely {
};
// egyszeres öröklődés: a személy osztályból származtatott osztály
class apa : public szemely {
};
```

```
// többszörös öröklődés: az anya és az apa osztályokból származtatott
// osztály
class gyermek : public anya, public apa {

};
```

A fenti példában definiált osztályhieracrhia grafikusan is ábrázolható:



Mint említettük, a származtatott osztály örökli az alaposztály(ok) tulajdonságait (mezőit), amelyek azonban meg is változtathatók:

- lehetőség van új tagok hozzáadására,
- a tagfüggvények újradefiniálására (*redefine*),
- az öröklött tagok elérhetőségének megváltoztatására.

A statikus osztályszerkezet a zártság tulajdonságának érvényesülése miatt nem minden esetben teszi lehetővé a tagfüggvények teljes átdefiniálását az származtatás során. Ekkor a többrétűség (*polymorphism*) mechanizmusát kell alkalmaznunk, amely virtuális (*virtual*) tagfüggvények formájában valósul meg a C++-ban. Az osztályhierarchián belül a virtuális tagfüggvények teljesen átdefiniálják az alaposztály ugyanilyen nevű és "kézjegyű" tagfüggvényét. Ez tehát azt jelenti, hogy attól függően, hogy programunk futása során mely szintjén vagyunk az objektum-hierarchiának, mindig más-más művelet (tagfüggvény) kerül végrehajtásra. Ily módon a program futása közben dől el, hogy végül is melyik tagfüggvényt kell aktivizálni. Ezt a jelenséget késői kötésnek (*late binding*) nevezzük, megkülönböztetésül a fordítás során megvalósított összerendeléstől (*early binding*).
A virtuális tagfüggvények használata alapvető követelmény minden objektum-könyvtártól.

Ha az *f2()* tagfüggvényt nem virtuális tagfüggvénynek deklaráljuk, akkor a zártság miatt a *bo.f1();* és az *uo.f1()* hívások hatására a program kimenet:

Alaposztály
Alaposztály

```
#include <iostream.h>

class alap {
public:
  void f1() { f2(); }
  void f2() { cout << "\n Alaposztály"; }
};

class  uj : public alap {
public:
  void f2() { cout << "\n Származtatott osztály"; }
};

void main()
{
 alap bo;
 uj   uo;
 bo.f1();
 uo.f1();
}
```

Az alaposztályban az *f2()* tagfüggvényt virtuálisnak definiálva

```
class alap {
public:
  void f1() { f2(); }
  virtual void f2() { cout << "\n Alaposztály"; }
};
```

a program kimenete a kívánt eredményt tartalmazza:

Alaposztály
Származtatott osztály

A továbbiakban az objektum-orientált C++ programozás eszközkészletével ismerkedünk meg.

F3.3.1 Osztályok definiálása a C++ nyelvben

A C++ **struct** deklaráció a C **struct** típus kiterjesztését tartalmazza. Ez a C **struct** típus minden tulajdonságával rendelkezik, azonban a kiterjesztéssel alkalmassá vált absztrakt adattípusok definiálására. A C++ rendelkezik egy új struktúrával is, a **class** (osztály) típussal.
A **struct** és a **class** egyaránt tartalmazhatnak adatmezőket (adattag), és ezekhez a mezőkhöz definiált műveleteket, függvénymezők (tagfüggvény) formájában. A **struct** és a **class** definíciók egyaránt használhatók osztályok definiálására, azonban a mezők alapértelmezés szerinti hozzáférésének következtében a **class** definíció áll közelebb az objektum-orientált gondolkodásmódhoz.

Az osztálydefiníció két részből áll. Az osztály feje a **class** alapszó után az osztály nevét tartalmazza. A másik rész az osztály teste, amelyet kapcsos zárójel fog közre és pontosvessző vagy objektumlista zár.

Általában az osztálydefiníció az adattagokon és tagfüggvényeken kívül a tagokhoz való hozzáférést szabályzó **public**, **private** és **protected** kulcsszavakat is tartalmazza:

```
class Test
{
   int num;
   float f;
 protected:
   char *ptr;
 public:
   void set (int i, float f, char *p);
   int getint(void);
   float getfloat(void);
};
```

F3.3.1.1 Adattagok

Az osztályban az adattagokat a C és a C++ változóihoz hasonlóan deklaráljuk, egyetlen különbség, hogy az adattagok nem tartalmazhatnak inicializációs listát. Ha egy osztályban valamely más osztály objektumát kívánjuk elhelyezni, akkor a másik osztály definíciójának meg kell előznie az aktuális osztály definícióját. A C++ mutatók és hivatkozások esetén megengedi az ún. előrevetett osztálydeklaráció használatát:


```
class Test1;    // előrevetett deklaráció
class Test2
{
 Test1 *to;
 ...
};
```

Osztályon belül nem lehet közvetlenül saját osztálydefinícióval rendelkező objektumot adattagként elhelyezni. Ehhez a másik osztályt előzőleg definiálni kell.

F3.3.1.2 Tagfüggvények

A tagfüggvényeket az osztály testén belül deklaráljuk. Ez a deklaráció vagy a függvény prototípusát vagy pedig az **inline** definícióját tartalmazza.

```
class Test
{
   int num;
   float f;
 protected:
   char *ptr;
 public:
   void set (int i, float f, char *p=0);  // prototípus
   int getint(void) { return num; }       // inline definíció
   float getfloat(void) { return f; }     // inline definíció
};
```

Az osztály deklarációját általában külön *include* file-ban helyezzük el, hogy elkerüljük a nem **inline** tagfüggvényeinek többszöri definícióját (TEST.H).

A nagyobb tagfüggvényeket az osztálydefiníción kívül adjuk meg, a :: operátor felhasználásával:

```
#include "test.h"
void Test::set(int i, float f, char *p)
{
 num=i;
 this->f=f;
 ptr=p;
}
```

A tagfüggvények több szempontból is eltérnek a hagyományos C++ függvényektől:

- Valamely osztály tagfüggvényei mindig elérik a saját osztály (**public**, **private**, **protected**) adattagjait. Nem tagfüggvény függvények csak a **public** adattagokhoz férhetnek hozzá (ha nem **friend** függvényei az osztálynak.)
- A tagfüggvények az osztály érvényességi tartományában definiáltak, ellentétben a közönséges függvényekkel, amelyek file szintű érvényességi körrel rendelkeznek.
- Tagfüggvényt csak ugyanazon osztálybeli tagfüggvénnyel lehet átdefiniálni, hiszen az átdefiniálás (*overloading*) mechanizmusa csak azonos érvényességi körrel rendelkező függvények között használható.

Fontos megértenünk az osztály adattagjainak és tagfüggvényeinek tárolását. Ha valamely osztálytípussal objektumot hozunk létre:

```
Test t1, t2;
```

akkor minden objektum saját adattagokkal rendelkezik (amennyiben az adattag nem statikus), és a tagfüggvények egyetlen példányát megosztva használják.

```
t1.set(1,3.5);
t2.set(2,5.6);
```

Felvetődik a kérdés, honnan tudja a *set()* függvény, hogy adott esetben mely adatterülettel kell dolgoznia? Erre a kérdésre a fordító nem látható tevékenysége adja meg a választ. Minden tagfüggvény, még a (**void**) is, rendelkezik egy nem látható paraméterrel (**this**), amelyben a fordító a hívás során az aktuális objektumra mutató pointert ad át. Továbbá, minden adattag hivatkozás automatikusan *this->adattag* hivatkozásként kerül be a kódba. A **this** (ez) mutató közvetlenül is felhasználható a tagfüggvényen belül. A előző példában a *this->f=f;* utasításban azért használtuk, mivel az adattag és a tagfüggvény paramétere ugyanazt a nevet *(f)* viseli.

F3.3.1.3 Az osztály tagjainak elérése

A C++ az osztály koncepcióban megvalósítja az információrejtés elvét. A **public** (közös), **private** (magán) és a **protected** (védett) alapszavakkal az egyes tagok elérését szabályozhatjuk.

```
class Clppp;
{
   // Itt helyezkednek el a private elérésű tagok.
   // Természetesen a private: is használható a tagok előtt megadva.
 protected:
   // A protected adatok és függvények deklarációja

 public:
   // A korlátozás nélkül elérhető adatok és függvények deklarációja
};
```

Az osztály deklarációjában több **public**, **private** és **protected** rész is elhelyezhető. A C++ 2.0 verziójától kezdődően a részeket **public, protected** és **private** sorrendben ajánlják elhelyezni:

```
class Test
{
 public:
   void set (int i, float f, char *p=0);/ // prototípus
   int getint(void) { return num; }       // inline definíció
   float getfloat(void) { return f; }     // inline definíció
 protected:
   char *ptr;
 private:
   int num;
   float f;
};
```

Nézzük meg az egyes előírások jelentését:

A **public** tag bárhonnan elérhető a programon belül, ahonnan maga az objektum elérhető. Az adatrejtés elvének érvényesüléséhez ajánlott, hogy **public** eléréssel csak tagfüggvényeket deklaráljunk.

A **protected** tagok külső függvények számára **private,** de a származtatott osztályok tagfüggvényei számára **public** elérésűek. Az osztálytagok **protected** elérése az osztályhierarchia kialakításánál, vagyis az öröklődésnél (*inheritance*) játszik szerepet.

A **private** tagokat csak az osztály saját tagfüggvényeiből, illetve az osztály "barátaiból" (***friend***) érhetjük el. A külső függvények és a leszármaztatott osztály tagfüggvényei (habár a **private** tagok is örökölődnek), nem rendelkeznek hozzáférési joggal a private osztálytagokhoz. Általában az osztály adattagjait **private** vagy **protected** eléréssel deklaráljuk.

F3.3.1.4 Az osztályok friend mechanizmusa

Vannak eseték, amikor az adatrejtési szabályok nem kívánt korlátozást jelentenek a programozó számára. A **friend** (barát) mechanizmus lehetővé teszi, hogy az osztály **private** és **protected** tagjait nem saját tagfüggvényből is elérjük. A **friend** deklarációt az osztály deklarációján belül kell elhelyeznünk. **Friend** lehet egy függvény, de akár egy egész osztály is:

```
class Fclass;
{
  friend Sclass;  // Az Sclass barát, így minden tagfüggvényére
                  // vonatkozik
                  // a baráti viszony. Ez a kapcsolat azonban nem
                  // automatikusan kölcsönös!

  friend int Tclass::count(int x);
                  // A Tclass osztálynak csak a count tagfüggvénye
                  // barát.

  friend long sum(void);
                  // A sum függvény barát.

                  // ...
};
```

F3.3.1.5 Az osztály objektumai

C++ nyelvben az objektumok az osztály példányait (*instance*) jelölik, amelyeket a szokásos változódefinícióval hozunk létre. A *Test* osztály példányait az alábbiak szerint hozhatjuk létre:

```
  Test      x;    // Test típusú objektum
  Test    *xp;    // mutató Test típusú objektumra
  Tesr & xr=x;    // referencia az x objektumhoz

  xp= new Test;   // az Test típusú objektum dinamikus létrehozása.
```


A függvényargumentumként használt objektum, illetve a függvényértékként definiált objektum érték szerint kerül átadásra. Ezekben az esetekben a fordító ideiglenes példányt készít az objektumról, ahova az adattagokat átmásolja. Az objektum másolása egyszerű értékadáskor is megtörténik:

```
Test t1, t2;
t2=t1;
```

A *t2=t1;* utasítás hatása megegyezik az alábbi 3 utasítás eredményével:

```
t2.num = t1.num;
t2.f   = t1.f;
t2.prt = t1.ptr;
```

Felhívjuk a figyelmet arra, hogy az értékadás után mindkét objektumban a *ptr* mutató ugyanazt a címet tartalmazza - amire a *t1.ptr* mutat, az nem másolódott át. Vannak esetek, amikor ez működés nem elfogadható.
A C++ lehetőséget biztosít arra, hogy az alapértelmezés szerinti másolási műveletet *(X::operátor=(const X&))* újradefiniáljuk. Hasonló a probléma akkor is, ha egy új objektumot már meglévővel inicializálunk:

```
Test a;
Test b=a;
```

Ekkor a meglévő objektum adattagjai szintén a fenti mechanizmus szerint másolódnak át az új objektum adattagjaiba. A másolást az ún. másoló (*copy*) konstruktor *(X::X(const &X))* végzi el, amelyet szintén újradefiniálhatunk.

Az osztályok tagjait a **struct** típusnál megismert operátorokkal érhetjük el. A pont (.) operátort statikus objektumok, míg a nyíl (->) operátort dinamikusan létrehozott objektumok esetén használjuk:

```
class BadClass
{
 public:
   int a;
   void init(int x) { a=x; }
};
void main()
{
  BadClass x;
  BadClass *px;
```

```
  px=new BadClass;
  x.a=26;              px->a=70;
  x.init(26);          px->init(70);
}
```

Az előzőekben már megemlítettük a **this** mutató szerepét. Most csak egy gondolat erejéig térünk vissza hozzá. Gyakori megoldás, hogy valamely tagfüggvénynek magát az objektumot vagy egy másolatot az objektumról kell visszaadnia.
Ekkor a **return** utasításban a **this** mutatót használjuk. A *return *this;* utasítás egy másolatot készít az objektumról és azzal tér vissza, míg a *return this;* magát az objektumot adja meg függvényértékként.

F3.3.1.6 Statikus osztálytagok használata

Érdekes felhasználási lehetőséget kínál a statikus adattagok definiálása az osztályban. A **static** adattagot (mint ahogy a tagfüggvényeket) megosztva használják az osztály objektumai. Ezzel elkerülhetjük azt a problémát, ami nem osztálytag globális változók használatából származhat.
A statikus adattag közvetlenül az osztályhoz tartozik, így az akkor is elérhető, ha egyetlen objektuma sem létezik az adott osztálynak.

A statikus adattag inicializálását az osztályon kívül kell elvégezni:

```
class Abc {
 static int def;
};
int Abc::def=5;
```

Az incializálás függetlenül a statikus adattag elérhetőségétől (**public**, **private**, **protected**), mindig elvégezhető. Ha a statikus adattag **public** elérésű, a programban bárhonnan hozzáférhetünk az osztály neve és a *scope* operátor felhasználásával (pl. *Abc::def*).

A statikus adattag kezelésére statikus tagfüggvény használhatunk. A statikus tagfügvénnyel normál adattagot nem érhetünk el, hiszen a paraméterei között nem szerepel a **this** mutató.


```
class Abc {
   static int def;
   int xyz;
 public:
  static void write(int x) { def=x;}
  static int read(void) { return def;}
};
int Abc::def=3;

void main()
{
  int x=Abc::read();
  Abc::write(12);
}
```

F3.3.1.7 Osztálytagokra mutató pointerek

C++-ban egy függvényre mutató pointer még akkor sem veheti fel valamely tagfüggvény címét, ha különben a típusuk és a paraméterlistájuk teljesen megegyezik. Ennek oka az, hogy a (nem statikus) tagfüggvények az osztály példányain fejtik ki hatásukat. Ugyanez igaz az adatmezőkhöz rendelt mutatók esetén.

A mutató helyes definiálásakor az osztály nevét és a *scope* operátort is használnunk kell:

```
int Tclass::*x;          // x mutató egy int típusú adattagra

int (Tclass::*fx)(void); // fx mutató egy olyan tagfüggvényre,
                         // amelyet argumentum nélkül hívunk
                        // és int értéket ad vissza
```

A következő példában az osztálytagokra mutató pointerek használatát mutatjuk be:

```
class PtrToClass
 {
  public:
    int a;
    void set(int x) { a=x;}
    int  get()      { return a;}
 };
```

```
void  main()
 {
   int   PtrToClass::*ip;            // Mindhárom tagot mutató
                                     // segítségével
   void (PtrToClass::*sp)(int);      // kívánjuk elérni
   int  (PtrToClass::*gp)(void);

   ip=&PtrToClass::a;              // A mutatók inicializálása
   sp= PtrToClass::set;
   gp= PtrToClass::get;

   PtrToClass o1,*o2;
   (o1.*sp)(12);                   // A o1.set meghívása
   (o1.*ip)++;                     // Az o1.a léptetése

   o2 =new PtrToClass;
   *o2=o1;
   (o2->*ip)*=11;                 //  Az o2->a szorzása 11-el
   int iv;
   iv=(o2->*gp)();                // Az o2->get meghívása
 }                                // Az iv változó értéke 143 lesz!
```

F3.3.2 Konstruktorok és destruktorok

Ebben a részben két speciális tagfüggvénnyel ismerkedünk meg, melyek feladata az osztályok inicializálása illetve deinicializálása.

F3.3.2.1 Az osztályobjektumok inicializálása

A programban használt változók és objektumok megfelelő inicializálása fontos feladat, hisz e lépés nélkül a program viselkedése véletlenszerűvé válhat. A C nyelvben megszokott struktúra-inicializálás a **class** típusú objektumok esetén akkor használható, ha az csak **public** elérésű adattagokkal rendelkezik. A C++ programokban az objektumok inicializálásának feladatát speciális tagfüggvényekkel a konstruktorokkal oldjuk meg.

F3.3.2.2 Konstruktorok

A konstruktor olyan speciális tagfüggvény, amely elvégzi az osztály objektumainak inicializalását. A konstruktor nevének meg kell egyeznie az őt tartalmazó osztály nevével. Az osztály konstruktorát a fordító minden olyan esetben automatikusan meghívja, amikor az adott osztály objektuma létrejön.

A konstruktor nem rendelkezik visszatérési értékkel, de különben ugyanúgy viselkedik mint bármely más tagfüggvény. Igy a konstruktor is átdefiniálható, így adott a lehetőség többféle inicializálás megvalósítására.
Fontos megértenünk, hogy a konstruktor nem foglal memóriát a létrejövő objektum számára - ezt a fordító végzi az alaptípusoknál használt módszerrel. (Ha az objektum dinamikus, akkor a **new** operátor foglal memóriát.) A konstruktor feladata a már lefoglalt memóriaterület inicializálását végzi el. Ha az objektum valamilyen mutatót tartalmaz, ami elég gyakori megoldás, akkor természetesen a konstruktorból kell gondoskodnunk a mutató által kijelölt terület létrehozásáról. A *Vector* osztályban több konstruktort is definiáltunk:

```
class Vector {
  public:                          // Példa az inicializálásra:
        Vector();                  // Vector a;
        Vector(int n);             // Vector a(12);
        Vector(Vector& v);         // Vector a(12), b=a;
        Vector(int a[],int n);     // int x[23]; Vector a(x,23);
 private:
        int * p;
        int size;
};
```

Ha a fenti osztálydefiníciót a VECTOR.H file-ban tároltuk, akkor a VECTOR.CPP file tartalmazhatja a tagfüggvények definícióját.

```
#include "vector.h"

// Argumentum nélküli konstruktor - (default kostruktor)
Vector::Vector(void)
{
    size = 10;          // Az alapértelmezés szerinti méret
    p = new int[ size];
}

// Adott méretű vektor inicilaizálása
Vector::Vector(int n)
{
    size = n;
    p = new int[ size];
}

// Inicializálás másik vektorral - Copy konstruktor
Vector::Vector(Vector& v)
{
     size = v.size;
     p = new[ size];
     // Az elemek átmásolása
     for (int i = 0; i < size; ++i)    p[ i] = v.p[ i];
}
```

```
// Inicializálás hagyományos n elemű tömbbel
Vector::Vector(int a[], int n)
{
    size = n;
    p = new int[size];
    for (int i = 0; i < size; ++i)  p[i] = a[i];
}
```

Mind a négy konstruktor helyet foglal a memóriából, a vektor elemi számára. Nézzünk néhány példát a *Vector* objektum definiálására.

```
// Az első (default) konstruktor működik - 10 elemű vektorok
   Vector a,b;              // minkét vektor 10 elemű
   Vector *c = new Vector; // c pointer egy 10 elemű vektorra
   Vector d[5];             // 5 darab 10 elemű vektor tömbje
   Vector *e=new Vector[5] // 5 darab 10 elemű vektor tömbje

// A második konstruktor aktivizálódik - 20 elemű vektorok
   Vector f = Vector (20); //
   Vector g(20);
   Vector h=20;
   Vector *i=new Vector(20);

// 3 elemű vektortömb létrehozása, melynek elemei 3, 7, illetve 9
// elemű vektorok:
   Vector j[]={3,7,9};

// A harmadik (másoló) konstruktor használata:
   Vector k;
   Vector l=k;
   Vecotr m(k);
   Vector n[3]={k,l,m};

// A negyedik konstruktor hívódik meg:
   int    z[20];
   Vector  p(z,20);
```

A konstruktor egyaránt lehet **public**, **private** és **protected** elérésű. A csak **private** konstruktorokat tartalmazó osztályt **private** osztálynak nevezzük. Ilyen osztály objektumait csak **friend** függvényben, illetve **friend** osztályban hozhatunk létre.
Az az osztály, amelyik csak **protected** konstruktorokat tartalmaz, és nem rendelkezik barátokkal (*friends*), jól használható absztrakt alaposztályként, amelyből más osztályokat származtathatunk.

F3.3.2.3 Destruktorok

A fenti vektor példában a konstruktorok memóriát foglalnak a vektor elemei számára, a **new** operátor felhasználásával. Ez a memória mindaddig lefoglalt marad, amíg fel nem szabadítjuk. Erre a célra a C++ biztosít egy speciális tagfüggvényt, a destruktort, amelyben gondoskodhatunk a lefoglalt terület felszabadításáról. A destruktor nevét a hullám karakterrel (~) egybeépített osztálynévként kell megadni. A destruktor, akárcsak a konstruktor, szintén nem rendelkezik visszatérési típussal.
A destruktort tartalmazó *Vector* osztály, amelyben a destruktor felszabadítja a konstruktor által lefoglalt memóriát, amikor az objektum megszűnik.

```
class Vector {
  public:                          // Példa az inicializálásra:
        Vector();                  // Vector a;
        Vector(int n);             // Vector a(12);
        Vector(Vector& v);         // Vector a(12), b=a;
        Vector(int a[],int n);     // int x[23]; Vector a(x,23);
        ~Vector() {delete p;}      // inline destruktor

 private:
        int * p;
        int size;
};
```

Ha az osztály rendelkezik destruktorral, a fordító minden olyan esetben meghívja azt, amikor az objektum érvényessége megszűnik. Kivétel a **new** operátorral létrehozott objektum, mely esetén a destruktor aktivizálása csak a delete operátor megadásával lehetséges. Szintén fontos megjegyeznünk, hogy a destruktor nem magát az objektumot szűnteti meg, hanem automatikusan elvégez néhány általunk definiált deinicializációs műveletet. (A destruktor explicit módon is meghívható.)

F3.3.2.4 Az objektum tagosztályainak inicializálása

Ha egy osztály tagként valamely másik osztály objektumát tartalmazza, akkor a tagosztály konstruktorának inicializálását a külső osztály konstruktorával oldjuk meg. Ez a mechanizmus tag-inicializációs lista használatát jelenti, amelyet a konstruktor után kettősponttal elválasztva adunk meg. A lista vesszővel elválasztott elemei az osztály tagjai, melyek után zárójelben áll az

inicializációs argumentum. A példában a *Kulso* osztályon belüli *Belso* típusú objektumot inicializáljuk a *Kulso* konstruktorában.

```
class Belso
{
    int cnt;
  public:
    Belso(int v) : cnt(v) {}
};

class Kulso
{
   int num;
   Belso obj;
  public:
   Kulso(int, int);
};

Kulso::Kulso(int k, int b) : obj(b)
{
  num=k;
}
```

A tag inicializációs listát csak a konstruktorok definíciójánál lehet megadni. Ezt a módszert tetszőleges típusú tag esetén is használhatjuk. A fenti konstruktor egy lehetséges másik alakja:

```
Kulso::Kulso(int k, int b) : obj(b), num(k)
{}
```

A tag-inicializációs lista használata kötelező, ha az osztály referencia típusú adattagot, vagy paraméterezett konstruktorral rendelkező objektumot tartalmaz. Az inicializációs lista feldolgozása a konstruktor törzsének végrehajtása előtt megy végbe.

F3.3.3 Operátorok átdefiniálása

A C++ nyelv biztosítja annak a lehetőségét, hogy valamely, programozó által definiált, függvényt szabványos operátorhoz kapcsoljunk, kibővítve ezzel az operátor működését. Ez a függvény automatikusan meghívódik, amikor az operátort egy meghatározott szövegkörnyezetben használjuk.

Az operátor-függvényt csak akkor definiálhatunk, ha annak legalább egy argumentuma osztály (**class, struct**) típusú. Ez azt jelenti, hogy a **void**

függvények, illetve a csak alap adattípusú argumentumokat használó függvények nem lehetnek operátor-függvények.

Az operátor-függvény deklarációjának formája:

```
return_típus  operator op(argumentum lista);
```

ahol az *op* helyén az alábbi C++ operátorok valamelyike állhat:

```
   []      ()      .      ->      ++      --
    &      *       +      -       ~       !
sizeof     /       %      <<      >>      <
    >      <=      >=     ==      !=      ^
    |      &&      !!     =       *=      /=
   %=      +=      -=     <<=     >>=     &=
   ^=      |=      ,                      ->*
```

Továbbra sem definiálhatók át a . , a .* , a :: és a ?: operátorok. Az operátorok átdefiniálásával az operátorok precedenciája nem változtatható meg, kivéve, ha zárójeleket használunk. Az operátor-függvényeket általában osztályon belül definiáljuk, a felhasználói típus lehetőségeink kiterjesztése céljából. Az =, (), [] és -> operátorokat csak nem statikus tagfüggvénnyel lehet átdefiniálni. A **new** és a **delete** operátorok esetén az átdefiniálás statikus tagfüggvénnyel történik. Minden más operátor-függvény megadható tagfüggvényként, vagy az osztály ***friend*** függvényeként.

Példaként tekintsük a *Vector* osztály kibővített változatát, amelyben az index ([]), az értékadás (=) és az összeadás (+) műveletét definiáltuk át:

```
// VECTOR.H
class Vector {
  public:
        Vector();
        Vector(int n);
        Vector(Vector& v);
        Vector(int a[],int n);
        ~Vector() { delete p; }
        int ub() { return size-1; }  // a felső indexhatár lekérdezése
        int& operator[] (int i);     // az [] operator átdefiniálása
        Vector& operator= (Vector& v); // az =  operator átdefiniálása
        Vector  operator+ (Vector& v); // a  +  operator átdefiniálása
  private:
        int * p;
        int size;
};
```

```
// VECTOR.CPP
#include <stdlib.h>
#include <iostream.h>
#include "vector.h"

int& Vector::operator[](int i)
{
        if (i < 0 || i > ub()) {
                cerr << "Illegális vektorindex:  " << i << "\n";
                exit(1);
        }
        return (p[i]);
}

Vector& Vector::operator=(Vector& v)
{
        int s = (size < v.size) ? size : v.size;
        for (int i = 0; i < s; ++i)  p[i] = v.p[i];
        return *this;
}

Vector Vector::operator+(Vector& v)
{
        int s = (size < v.size) ? size : v.size;
        Vector sum(s);
        for (int i = 0; i < s; ++i)  sum.p[i] = p[i] + v.p[i];
        return (sum);
}
```

A szabványos C++ operátorokat két csoportra oszthatjuk, az operandusok száma alapján. Erre a két esetre az alábbi táblázatban foglaltuk össze az átdefiniált operátor és a tagfüggvények kapcsolatát.

Kétoperandusú operátor esetén:

Tag állapot	Szintaxis	Aktuális hívás
tagfüggvény	`X op Y`	`X.operator op(Y)`
friend	`X op Y`	`operator op(X,Y)`

Egyoperandusú operátor esetén:

Tag állapot	Szintaxis	Aktuális hívás
tagfüggvény	`op X`	`X.operator op()`
tagfüggvény	`X op`	`X.operator op()`
friend	`op X`	`operator op(X)`
friend	`X op`	`operator op(X)`

F3.3.3.1 A new és a delete operátorok átdefiniálása

A **new** és a ***delete*** operátorok átdefiniálásakor további megkötéseket kell szem előtt tartanunk.
A **new** operátor osztálytag operátor-függvényét **void *** visszatérési értékkel és **size_t** típusú első argumentummal kell definiálni. Ebben az argumentumban a fordító automatikusan átadja az osztály byte-ban kifejezett méretét.

A *delete* operátor esetén, az osztálytag operátor-függvényt **void *** első és a **sizet_t** típusú opcionális második argumentummal kell definiálnunk. A második argumentumot, amennyiben megadtuk, a fordító használja az első argumentum által megcímzett objektum méretének átadására.

Az átdefiniált **new** és **delete** operátorok mellett, a érvényességi kör operátorának megadásával mindig elérhetjük az eredeti ***new*** és **delete** operátorokat:

```
::delete p;
```

A **new** és ***delete*** operátorok automatikusan az őket definiáló osztály statikus tagjaként kerülnek lefordításra. Ebből következik, hogy a this mutató nem használható, tehát az operátor-függvényekből csak a statikus adattagokat érhetjük el. Ez azért szükséges, mivel a **new** és **delete** meghívását az osztályobjektum létezése nélkül is végre kell hajtani.

F3.3.3.2 Felhasználó által definiált típuskonverzió

A C++ nyelv támogatja, hogy a programozó a saját adattípusához (osztály) típuskonverziókat rendeljen hozzá. Az ilyen felhasználó által definiált típuskonverziót végrehajtó tagfüggvény deklarációja:

```
operator típus();
```

A függvény visszatérési értéke automatikusan a függvény neveként használt típussal egyezik meg. A típuskonverziós operátor csak visszatérési típus és argumentumlista nélküli tagfüggvény lehet.

Az alábbi példában a komplex típust osztályként valósítjuk meg. Az egyetlen nem *cplx* típusú argumentummal rendelkező konstruktor elvégzi a más típusról - a példában **double** - *cplx* típusra történő konverziót. A fordított irányú konverzióhoz **double** nevű konverziós operátort definiáltunk.

```
#include <math.h>
#include <iostream.h>

class cplx
{
 public:
   cpl   // konverziós konstruktor
x(double a) { re=a; im=0; }
   cplx(double a, double b) { re=a; im=b;}
   // konverziós operátor
   operator double() { return sqrt(re*re+im*im);}
 private:
  double re, im;
};

void main()
{
   cplx a(3,4);
   cout << double(a)<< "\n";    // a kiírt érték:  5
   cout <<double(cplx(10));     // a kiírt érték: 10
}
```


F3.3.4 Az öröklődés mechanizmusa

Az öröklődés (*inheritance*) a C++ nyelv legfőbb sajátossága. Ez a mechanizmus teszi lehetővé, hogy bizonyos osztályokból más osztályokat származtatsunk, mely osztályok a származtatás során adattagokat és tagfüggvényeket örökölnek. Az öröklött tulajdonságok tetszőlegesen kiterjeszthetők és megváltoztathatók.

A C++ 2.0 változata támogatja a többszörös öröklődést (*multiple inheritance*), melynek során egy új osztályt több alaposztályból származtatunk.

F3.3.4.1 A származtatott osztályok

A származtatott osztály (*derived class*) olyan osztály, amely az adattagjait és a tagfüggvényeit egy vagy több előzőleg definiált osztálytól örökli. Azt az osztályt, amelytől a származtatott osztály örököl, alaposztálynak (*base class*) nevezzük. A származtatott osztály szintén lehet alaposztálya további osztályoknak, lehetővé téve ezzel az osztályhierarchia kialakítását.

A származtatott osztály az alaposztály minden tagját örökli, de az alaposztályból csak a **public** és **protected** tagokat éri el sajátjaiként. A származtatott osztály az öröklött tagokat saját adattagokkal és tagfüggvényekkel egészítheti ki.

Az származtatás kijelölésére az osztály fejét használjuk:

```
class  Derived : public Base1, ...private BaseN
{
 // az osztály törzse
}
```

A származtatási listában megadott **public** és **private** kulcsszavak az öröklött tagok elérhetőségét szabályozzák. A **public** származtatás során az öröklött tagok megtartják az alaposztálybeli állapotukat, míg a **private** származtatás során az öröklött tagok **private** tagokká válnak. (Az alapértelmezés szerinti származtatási mód **class** típusú alaposztály esetén a **private**, míg a **struct** típust használva a **public.**)

Fontos megjegyeznünk, hogy a **public** származtatással létrehozott osztály minden esetben (értékadás, függvényargumentum,...) helyettesítheti az alaposztályt. Ezen nem kell csodálkozni, hisz a származtatott osztály magában foglalja az alaposztályt.

A friend viszony az öröklődés során

Az alaposztály barátja (**friend**) a származtatott osztályban csak az alaposztályból öröklött tagokat érheti el. A származtatott osztály barátja (**friend**) az alaposztályból csak a **public** és a statikus **protected** tagokat érheti el.

Az öröklött adattagok elérése

Általában a származtatott osztály öröklött tagjai ugyanúgy érhetők el, mint a saját tagok. Azonban elképzelhető olyan eset, amikor az öröklött tagok elérése akadályba ütközik. Ha származtatott osztály azonos névvel újradefiniálja az öröklött tagot, az láthatatlanná válik. Ilyen esetben a osztály érvényességi kör operátort kell használnunk: *Base::tag*

Nézzünk egy példát az elmondottak szemléltetésére.

```
class B                  // az alaposztály
{
   int x,y;
 public:
   int b1,b2;
   int Bfunc1(void) { return x;}
   int Bfunc2(void) { return y;}
};

class D: private B      // a származtatott osztály
{
   int b2;             // az öröklött b2 újradefiniálása
   int d;
 public:
    B::b1;             // a private származtatással öröklött
                       // b1 public elérésű lesz
   void Bfunc1(void); // az öröklött Bfunc1 újradefiniálása
};

void D::Bfunc1(void)
{
     d =B::Bfunc1(); // a nem látható Bfunc1() elérése
     b1=Bfunc2();    // a Bfunc2() látható
     b2=B::b2;       // a nem látható b2 adattag elérése
}
```



```
void main()
{
 D od;
 od.b1=13;
}
```

A B alaposztály tagjai:

```
private: x,y
public:  b1, b2, Bfunc1(), Bfunc2()
```

A D származtatott osztály tagjai:

```
private: B::b2, b2, d, B::Bfunc1(), Bfunc2()
public:  b1, Bfunc1()
```

F3.3.4.2 Az alaposztály inicializálása

Az alaposztály(ok) inicializálására a tag-inicializációs lista kibővített változatát használjuk. Példaként tekintsük aa *base1* és *base2* osztályokból származtatott *Derived* osztály konstruktorát:

```
Derived::Derived(int ct, char *ptr, long val, char st,
         short tp) : base1(ct,val), base2(tp,ptr)
{
 // a származtatott osztály konstruktora
}
```

Az alaposztályok konstuktorai az inicializációs lista szerinti sorrendben hívódnak meg, balról jobbra haladva. Először az alaposztályok konstruktorai kerülnek végrehajtásra, majd a tagosztályok konstruktorai következnek (ha vannak) és végül a sort a származtatott osztály konstruktorának végrehajtása zárja. Az alaposztály inicializálása elvégezhető alaposztály objektummal, vagy a származtatott osztály objektumával, ha a származtatást **public** módon végeztük. Az inicializálást a másoló konstruktor végzi (*X::X(const &X)*):

```
Derived::Derived(int ct, Derived& dc) : Base(dc)
{
 // a származtatott osztály konstruktora
}
```

F3.3.4.3 Virtuális függvények

A virtuális függvény olyan **public** vagy **protected** tagfüggvénye a **public** alaposztálynak, amelyet a származtatott osztályban újradefiniálhatunk az osztály tulajdonságainak megváltoztatása érdekében. A virtuális függvény általában a **public** alaposztály referenciáján vagy mutatóján keresztül kerül meghívásra, melynek aktuális értéke a program futása során alakul ki (dinamikus kapcsolás).

Ahhoz, hogy egy tagfüggvény virtuális legyen a **virtual** alapszót kell használnunk a függvény deklarációja előtt:

```
class Temp
{
  public:
    virtual int dof();
};
```

Nem szükséges, hogy az alaposztályban a virtuális függvénynek a definíciója is szerepjen. Ebben az esetben ún. tiszta virtuális függvénnyel (*pure virtual function*) van dolgunk:

```
class Temp
{
  public:
    virtual int dof()=0;
};
```

Egy vagy több tiszta virtuális függvényt tartalmazó alaposztály (absztrakt osztály) nem hozhat létre objektum példányt. Az absztrakt osztály csak alaposztályként használható.

A virtuális függvények újradefiniálása

Ha egy függvényt az alaposztályban virtuálisként deklarálunk, akkor ezt a tulajdonságát az öröklődés során is megőrzi. A származtatott osztályban a virtuális függvényt saját változattal újradefiniálhatjuk, de az öröklött verziót is használhatjuk. Saját verzió definiálásakor nem szükséges a **virtual** szót megadnunk.

Ha egy származtatott osztály tiszta virtuális függvényt örököl, akkor ezt mindenképpen saját verzióval kell újradefiniálni, mivel különben a származtatott osztály is absztrakt osztály lesz. A származtatott osztály tartalmazhat olyan virtuális függvényeket is, amelyeket nem az alaposztálytól örökölt.

A származtatott osztályban a virtuális függvény újradefiniált változatának pontosan (név, típus, argumentumlista) meg kell egyeznie az alaposztályban definiálttal. Ha a két deklaráció nem egyezik pontosan, akkor az újradefiniálás helyett az átdefiniálás mechanizmusa érvényesül.

F3.3.4.4 Virtuális destruktorok

A destruktort virtuális függvényként is definiálhatjuk. Ha az alaposztály destruktora virtuális, akkor minden ebből származtatott osztály destruktora is virtuális lesz. Ezáltal biztosak lehetünk abban, hogy a megfelelő destruktor kerül meghívásra, amikor a **delete** operátorral az objektumot megszüntetjük.

F3.3.4.5 Virtuális alaposztályok

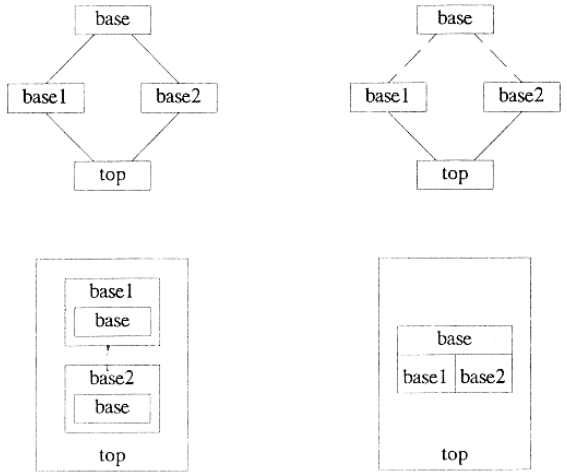
A többszörös öröklődés során problémát jelenthet, ha ugyanazon alaposztály több példányban jelenik meg a származtatott osztályban. A virtuális alaposztályok használatával az ilyen jellegű problémák kiküszöbölhetők.

```
#include <iostream.h>

class base
{
 public:
  int q;
};

class base1 : virtual public base
{
   char x;
 public:
   base1(int i)  { x=i;}
};
```

```
class base2:  public virtual base
{
   long y;
  public:
   base2(int i):y(i) {}
};
class top:  public base1,  public base2
{
   int a,b;
  public:
   top(int i,int j):base1(i*5),base2(j+i), a(i)  {b=j;}
};
void main()
{
  top to(1,26);
  to.base1::q=70;
  cout << to.base2::q;     // 70 íródik ki
}
```



A base nem virtuális alaposztály

A base virtuális alaposztály

Az alaposztály a **virtual** definíció hatására csak egyetlen példányban lesz jelen a származtatott osztályokban, függetlenül attól, hogy hányszor fordul elő az öröklődési láncban. A példában a virtuális alaposztály *q* adattagját örökli a *base1* és *base2* alaposztályok is.
A virtualitás miatt a *base* bázisosztály egyetlen példányban szerepel, így a *base1::q* és a *base2::q* ugyanarra az adattagra hivatkozik. A **virtual** szó használata nélkül a *base1::q* és a *base2::q* különböző adattagokat jelölnek.

F3.4 Paraméterezett típusok (templates)

A sablonok (*template*) használata lehetővé teszi, hogy egymással kapcsolatban álló függvények és osztályok családját hozzuk létre.

F3.4.1 Paraméterezett függvények

Tegyük fel, hogy olyan függvényt akarunk készíteni, amely két érték közül kiválasztja a maximálist. Ahhoz, hogy minden típus esetén működjön ez a függvény, használni kell az átdefiniálás mechanizmusát. Az átdefiniálás során a függvények törzse gyakorlatilag nem változik, így teljesen logikus a paraméterezett függvények használata.

Paraméterezett függvények esetén az általunk definiált általánosított függvényből maga a fordító program állítja elő a szükséges formáját a hivatkozott függvényeknek.

```
#include <iostream.h>

// paraméterezett függvény definíciója
template<class ANY_TYPE>
ANY_TYPE maximum(ANY_TYPE a, ANY_TYPE b)
{
   return (a > b) ? a : b;
}

void main()
{
  int x = 12, y = -7;
  float real = 3.1415;
  char ch = 'A';

   cout << maximum(x, y)    << endl;
   cout << maximum(-34, y) << endl;
   cout << maximum(real, float(y)) << endl;
   cout << maximum(real, float(x)) << endl;
   cout << maximum(ch, 'X') << endl;
}
```

F3.4.2 Paraméterezett osztályok

Az paraméterezett osztály (*generic class*, vagy *class generator*), lehetővé teszi, hogy más osztályok definiálásához a paraméterezett osztályt, mint mintát használjuk. Ezáltal egy adott osztálydefiníció minden típus esetén alkalmazható.
A példánkban a *stack* paraméterezett osztályt használjuk, különböző típusú elemek tárolására.

```
#include <iostream.h>

const int MAXSIZE = 128;

// A paraméterezett osztály definíciója
template<class ANY_TYPE>
class stack
{
   ANY_TYPE array[MAXSIZE];
   int stack_pointer;
public:
   stack(void) { stack_pointer = 0; };
   void push(ANY_TYPE in_dat) { array[stack_pointer++] = in_dat; };
   ANY_TYPE pop(void)     { return array[--stack_pointer]; };
   int empty(void)        { return (stack_pointer == 0); };
}

char name[] = "Borland C++ 3.0 / 3.1";

void main(void)
{
  int   x = 12, y = -7;
  float real = 3.1415;

  stack<int> int_stack;              // int stack
  stack<float> float_stack;          // float stack
  stack<char *> string_stack;        // string stack

  int_stack.push(x);                 // int stack feltöltése
  int_stack.push(y);
  int_stack.push(77);

  float_stack.push(real);           // float stack töltése
  float_stack.push(-12.35);
  float_stack.push(100.0);

   string_stack.push("1. Line");    // a string stack töltése
   string_stack.push("2. Line");
   string_stack.push("3. Line");
   string_stack.push(name);
```

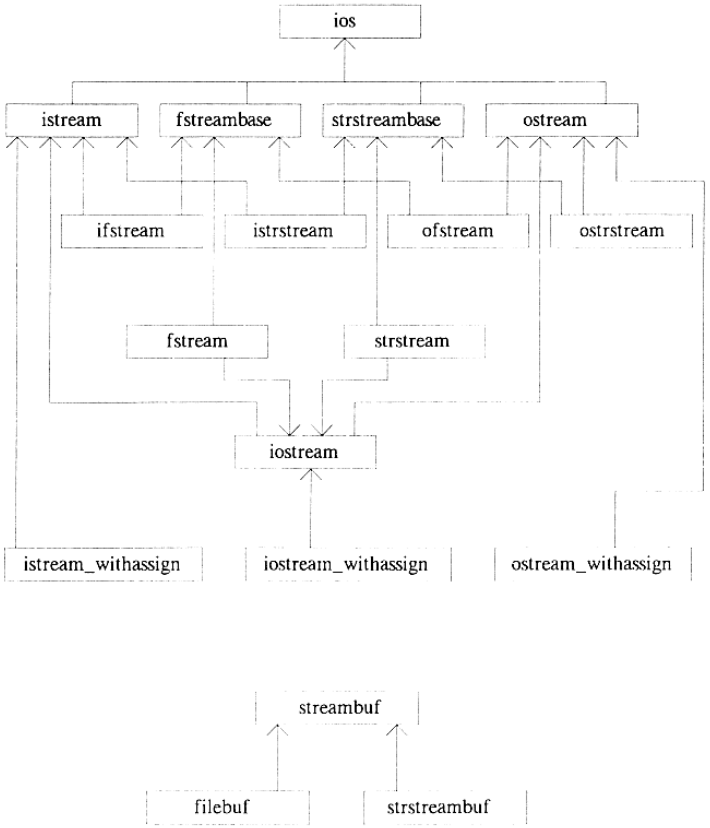


```
   cout << "\nInteger stack ---> ";  // az int stack kiolvasása
   cout << "\t" << int_stack.pop();
   cout << "\t" << int_stack.pop();
   cout << "\t" << int_stack.pop();

   cout << "\nFloat stack ---> ";  // a float stack kiolvasása
   cout << "\t" << float_stack.pop();
   cout << "\t" << float_stack.pop();
   cout << "\t" << float_stack.pop();

   cout << "\n\n   Strings";      // a string stack kiolvasása
   do {
      cout << "\n" << string_stack.pop();
   } while (!string_stack.empty());
}
```

F3.5 A C++ stream input/output



A C++ I/O stream-osztályok hierarchiája

Minden programban fontos szerepet játszanak az input/output műveletek. A C nyelvben szabványos könyvtári függvények szolgálnak ezen feladatok elvégzésére. A C++ nyelvben a *stream* osztályok hierarchiáján keresztül valósulnak meg az I/O műveletek.

Az **iostream** könyvtár két párhuzamos osztályra, a **streambuf** és az **ios** osztályokra épül. A **streambuf** osztály általános eljárásokat tartalmaz a pufferezésre és *stream*-kezelésére. Az **ios** osztály mutatót tartalmaz a **streambuf** osztályra. A különböző származtatott osztályok részletes ismertetése nélkül betekintést nyújtunk a *stream*-ek használatába.

F3.5.1 Szabványos stream-ek

Az IOSTREAM.H tartalmazza a négy megszokott C stream C++ objektumainak definícióját:

cin	*a szabványos input,*
cout	*a szabványos output,*
cerr	*a szabványos hiba egység (nem pufferezett),*
clog	*a **cerr** teljesen pufferezett változata.*

A *stream*-ek a << és a >> operátorok átdefiniált változataival valósítják meg az I/O műveleteket:

```
#include <iostream.h>

void main()
{
  int a,b;
  cout << "A és B : ";
  cin >> a >> b;
  cout << a << "  " << b << endl;
}
```

F3.5.2 Adatkivitel

Az adatkivitel művelete az **ostream** osztály tagfüggvényeinek felhasználásával a programozó igényének megfelelően paraméterezhető. A **put** tagfüggvénnyel egyetlen karakter írhatunk ki, míg a **write** tagfüggvény nagyméretű adatterületek kiírását támogatja. A **width** tagfüggvény segítségével a kiírás mezőszélességét definiálhatjuk.

Egyszerűbb megoldást jelent a különböző formátumváltozók beállítására a manipulátorok használata. A manipulátorok (**dec**, **hex**, **endl**, **setfill**, ...)

segítségével konverziókat és különböző formátummal kapcsolatos műveleteket hajthatunk végre. Például a hexadecimális kiíráshoz a **hex** manipulátort kell megadnunk:

```
cout << hex << 1970 ;
```

Szintén beállítható a szóköz karaktert helyettesítő kitöltő karakter (**fill**) A kiírt szöveget pedig a **setf** tagfüggvény segítségével igazíthatjuk ki:

```
cout.setf(ios::left, ios::adjustfiled);
```

A **setf** és a **unsetf** tagfüggvények a felhasználásával más formátumadatok is hasonlóan megváltoztathatók.

F3.5.3 Adatbeolvasás

Az adatbevitelre használt **istream** osztály az **ostream** osztályhoz hasonlóan, tagfüggvények sorozatát tartalmazza:

get	karakter beolvasása,
read	a **ostream write** tagfüggvényének párja,
putback	egy adott karaktert visszaír a pufferbe.

F3.5.4 A felhasználói típusok I/O műveletei

A *stream* input (>>) és output (<<) operátorait **friend** operátor-függvénnyel definiáljuk át.

```
#include <iostream.h>
#include <stdio.h>

class complex
  {
     double re,im;

   public:
     complex( double x=0, double y=0 )   { re=x; im=y; } ;
     friend   ostream& operator << ( ostream&, complex ) ;
     friend   istream& operator >> ( istream&, complex& ) ;
  } ;
```



```
ostream& operator << ( ostream& ost, complex com )
{
 ost<<com.re ;
 if ( com.im<0 ) ost<<" - i"<<-com.im ;
 else ost<<" + i"<<com.im ;
 return ost;
}

istream& operator >> ( istream& ist, complex& com )
{
 char cin[ 80] ;
 char c ;

 ist>>cin;
 sscanf(cin, "%lf%ci%lf", &com.re, &c, &com.im );
 if ( c == '-' )    com.im = - com.im ;
 return ist;
}

void main()
{
 complex a(4,5),c;
 cout <<"a="<< a<<"\n";
 cin>>c;
 cout <<"c="<< c<<"\n";
}
```

Ahhoz, hogy az I/O operátorokat láncba lehessen fűzni, a operátor-függvényekben az első argumentumként kapott *stream*-et függvényértékként adjuk vissza.

F3.5.5 Egyszerű file I/O

Az **ofstream** és az **ifstream** osztályok rendelkeznek a file-kezeléshez szükséges konstruktorokkal és tagfüggvényekkel. Az osztályok használata esetén az FSTREAM.H file-t kell a programunkba beépíteni.

A file-kezelés szokásos lépéseit az alábbi példaprogrammal kívánjuk bemutatni:

```
#include <iostream.h>
#include <process.h>
#include <fstream.h>

main(int argc, char * argv[])
{
  char ch;
  if (argc!=3)
    {
     cerr<<"Használat: dcopy file-ból file-ba"<<endl;
     exit(-1);
    }

  ifstream src;
  ofstream dest;

  src.open(argv[1],ios::nocreate|ios::binary);
  if (src.fail())
    {
     cerr<<"A "<<argv[1]<<" file nem nyitható meg olvasásra!"<<endl;
     exit(-1);
    }

  dest.open(argv[2],ios::binary);
  if (!dest)
    {
     cerr<<"A "<<argv[2]<<" file nem nyitható meg írásra!"<<endl;
     exit(-1);
    }

  while (dest && src.get(ch))     dest.put(ch);

  cout << "A másolás véget ért!"<<endl;

  src.close();

  dest.flush();
  dest.close();
  return 0;
}
```