TÍPUSOK
•Méret: implementáció függő
•A C++ objektumainak mérete mindig a char méretének többszöröse, így a char mérete 1. Egy objektum méretét a sizeof operátorral kapjuk meg.
1≡sizeof(char)≤sizeof(short)≤sizeof(int)≤sizeof(long)
1≤sizeof(bool) ≤sizeof(long)
sizeof(char) ≤sizeof(wchar_t) ≤sizeof(long)
sizeof(float) ≤sizeof(double) ≤sizeof(long double)
sizeof(char)<=sizeof(int)
sizeof(bool)<=sizeof(long)
sizeof(N)≡sizeof(signed N)≡sizeof(unsigned N) (N lehet char, short int, int vagy long int)
ÉLETTARTAM (L: létrejön, M: megszűnik)
•Automatikus változók: stack
L: amikor a vezérlés eléri a definiciót
M: amikor a definiciós blokkot elhagyjuk
•Lokális statikus változók: statikus tárterület
L: amikor a vezérés eléri a definiciót
M: a program futásának végén
•Globális/névtérbeli/osztálystatikus változók: statikus tárterület
L: program indulásakor
M: a program futásának végén
•Dinamikus változók: heap
L: new hívásakor
M: delete hívásakor
(int p=new int(3); delete p; int p=new int[size]; delete[] p;)
•Tömbelem:
L: a tömb létrejöttekor
M: a tömb megszűnésekor
•Osztály adattag:
L: a tartalmazó objektum létrejöttekor
M: a tartalmazó objektum megszűnésekor
•Temporális objektumok
L: részkifejezés kiértékelésekor
M: amint kiértékelődött a teljes kifejezés
DEKLARÁCIÓ, DEFINICIÓ
•A C++ programokban minden név számára mindig pontosan egy definició (meghatározás) létezhet, ugyanakkor a nevet többször is deklarálhatunk (bevezethetjük). Egy egyed minden deklarációja meg kell, hogy egyezzen a hivatkozott egyed típusában.
•A deklarációk négy részből állnak: egy nem kötelező minősítőből, egy alaptípusból, egy deklarátorból és egy – szintén nem kötelező – kezdőérték-adó kifejezésből.
-minősítő (specifier): egy kulcsszó, pl. a virtual, extern.
-deklarátor: egy névből és néhány nem kötelező operátorból áll (pl.: előtag: * mutató, *const konstans mutató, & referencia; utótag: [] tömb, () fv)
•def: int x; char* c; int getHeight(){ } (a fordító „tudja” mekkora helyet kell foglalni neki)
•dekl: extern int x; class car; int f(); (a nevet összekapcsolja egy típussal, a nevet bevezeti egy hatókörbe)
•extern: más fordítási egységben definiált objektum deklarációja
•typedef: az a deklaráció, amit a typedef kulcsszó előz meg, a típus számára új nevet hoz létre, nem egy adott típusú változót!
MUTATÓK
•Ha T egy típus, a T* a „T-re hivatkozó mutató” típus lesz, azaz egy T* típusú változó egy T típusú objektum címét tartalmazhatja.
char c=’a’;
char* p=&c //a p a c címét tárolja
•A mutatón végezhető alapvető művelet a „dereferencia”, azaz a mutató által mutatott objektumra való hivatkozás. (jele: az előtagként használt egyoperandusú *)
char c=’a’;
char* p=&c //a p a c címét tárolja
char c2=*p; //c2==’a’
•A tömbelemekre hivatkozó mutatókon aritmetikai műveleteket is végezhetünk.

TÖMBÖK

•Ha T egy típus, a T[size] a „size db T típusú elemből álló tömb” típus lesz.

float v[3]; char* a[32];

többdimenziós tömbök: int d2[10][20];

tömbök feltöltése:

int v1[]={1,2,3,4};

int v1[8]={1,2,3,4}; //=…{1,2,3,4,0,0,0,0}

KARAKTERLITERÁLOK

•A karakterliterál egy macskakörmökkel határolt karaktersorozat:

egy karakterliterál a ’\0’ null karakterre végződik

•sizeof(”Hello”)==6, ”Hello” típusa const char[6]

TÖMBÖKRE HIVATKOZÓ MUTATÓK

int v[ ] = { 1, 2, 3, 4 };

int* p1 = v; // mutató a kezdőelemre (automatikus konverzió)

int* p2 = &v[0]; // mutató a kezdőelemre

int* p3 = &v[4]; // mutató az "utolsó utáni" elemre

KONSTANSOK

•A const kulcsszó hozzáadható egy objektum deklarációjához, jelezve, hogy az objektumot állandóként határozzuk meg. Mivel egy állandónak később nem lehet értéket adni, kezdeti értékadást kell végeznünk.

•Ha valamit const-ként határozunk meg, az biztosíték arra, hogy hatókörén belül értéke nem fog megváltozni.

MUTATÓK ÉS KONSTANSOK

•A mutatók használatakor két objektummal kapcsolatos dologról van szó: magáról a mutatóról és az általa mutatott objektumról. Ha a mutató deklarációját a const szó előzi meg, akkor az az objektumot, és nem a mutatót határozza meg állandóként. Ahhoz, hogy állandóként egy mutatót, és ne az általa mutatott objektumot vezessük be, a *const deklarátort kell használnunk a sima * helyett.

```
void f1(char* p)
{   char s[ ] = "Gorm";
    const char* pc = s;          // mutató állandóra
    pc[3] = 'g';                 // hiba: pc állandóra mutat
    pc = p;                      // rendben
    char *const cp = s;          // konstans mutató
    cp[3] = 'a';                 // rendben
    cp = p;                      // hiba: cp konstans
    const char *const cpc = s; // konstans mutató állandóra
    cpc[3] = 'a';                // hiba: cpc állandóra mutat
    cpc = p;                     // hiba: cpc konstans
}
```

•Segítséget jelent, ha jobbról balra olvassuk ki.

const int* pi1=…; //mutató állandóra: pointer módosítható (átállítható), a mutatott érték nem

int* const pi2; //konstans mutató: pointer nem módosítható, a mutatott érték igen

const int* const pi3=…; //konstans mutató állandóra: egyik sem módosítható

REFERENCIÁK

•A referencia (hivatkozás) egy objektum „álneve” (alias). Az ilyen hivatkozásokat általában függvények és különösen túlterhelt operátorok paramétereinek és visszatérési értékeinek megadására használjuk. Az X& jelölés jelentése „referencia X-re”.

```
int i = 1;
int& r = i;    // r és i itt ugyanarra az int-re hivatkoznak
int x = r;     // x = 1
r = 2;         // i = 2
```

•a hivatkozás célpontját már létrehozáskor meg kell határoznunk

•a referencián nem hajtódik végre egyetlen művelet sem

```
int ii = 0;
int& rr = ii;
rr++;              // ii növelése eggyel
int* pp = &rr;  // pp az ii-re mutat
```

•a referenciák értéke már nem módosítható a kezdeti értékadás után

• A referenciákat olyan függvényparaméterek megadására is használhatjuk, melyeken keresztül a függvény módosíthatja a neki átadott objektum értékét

•A paraméterátadás a kezdeti értékadáshoz hasonló. Ha azt szeretnénk, hogy a program olvasható maradjon, legjobb, ha elkerüljük az olyan függvényeket, amelyek módosítják paramétereiket. Ehelyett meghatározhatjuk a függvény által visszaadandó értéket vagy mutató paramétert adhatunk
neki:
int next(int p) { return p+1; }
void incr(int* p) { (*p)++; }

FV-EK

•Az önmagukat meghívó függvényeket rekurzív (újrahívó) függvényeknek nevezzük.
int fac(int n) { return (n>1) ? n*fac(n-1) : 1; }
•Túlterhelés: A fordítóprogram a túlterhelt függvények halmazából úgy választja ki a megfelelő változatot, hogy megkeresi azt a függvényt, amelyiknél a hívás paraméter-kifejezésének típusa a legjobban illeszkedik a függvény formális paramétereire
•inline: javaslat a függvény kódjának beillesztésére a hívások helyein (-: nagyobb méretű kód; +: fv hívás megspórolása)

MAKRÓK

preprocesszor direktívák:
#define; #undef; #ifdef; #endif; #if; #include; #elif; #else; #ifndef; #line; #error; #pragma, …

OSZTÁLYOK

•**hozzáférés-meghatározás**: class alapértelmezetten private ;struct alapértelmezetten public
private
protected
public
•**konstruktor**t arról ismerjük meg, hogy ugyanaz a neve, mint magának az osztálynak, ha egy osztály rendelkezik konstruktorral, akkor minden, ebbe az osztályba tartozó objektum kap kezdőértéket. A konstruktornak adhatunk meg paramétereket is.
• A konstruktorokra ugyanazok a **túlterhelés**i szabályok vonatkoznak, mint más függvényekre
•Az olyan változókat, melyek egy osztályhoz tartoznak, de annak objektumaihoz nem, **statikus tag**nak nevezzük. A statikus tagokból mindig pontosan egy példány létezik nem pedig objektumonként egy, mint a közönséges, nem statikus adattagokból
•Az olyan függvényeket, melyek egy adott osztálytaghoz hozzáférnek, de nem szükséges objektumra meghívni azokat, **statikus tagfüggvény**nek hívjuk
•Alapértelmezés szerint az osztály típusú objektumok másolhatók és kezdőértékként egy azonos típusú osztály egy objektumának másolatát is kaphatják, még akkor is, ha konstruktorokat is megadtunk
• Alapértelmezés szerint az osztály objektum másolata minden tag másolatából áll. Ha nem ez a megfelelő viselkedés egy X osztály számára, az X::X(const X&) **másoló konstruktor**ral megváltoztathatjuk azt.
•másoló konstruktor hívódik: változó kezdeti értékének beállításakor, nem-referencia függvényparaméter átadásakor, függvény nem-referenciával visszatérésekor, kivétel dobásánál és elkapásánál.
•**konstans tagfüggvények**: a const minősítő a függvénydeklarációban (a paraméterlista után) azt jelenti, hogy ez a függvény nem változtatja meg az objektum állapotát.
• **Egy konstans tagfüggvényt alkalmazhatunk állandó (konstans) és változó (nem konstans) objektumokra is, a nem konstans tagfüggvényeket viszont csak nem konstans objektumokra**
• Minden (nem statikus) tagfüggvény tudja, melyik objektumra hívták meg, így pontosan hivatkozhat rá (pl. return ***this**;); A *this kifejezés azt az objektumot jelenti, amelyre a tagfüggvényt meghívták. Egy nem statikus tagfüggvényben a this kulcsszó egy mutatót jelent arra az objektumra, amelyre a függvényt meghívták.
•**Operátorok túlterhelése**: pl.: operator==, operator() stb. megadjuk, hogy az adott osztály hogyan használja
•**Destruktorok:** Az objektumok kezdőállapotát a konstruktorok állítják be, vagyis a konstruktorok hozzák létre azt a környezetet, amelyben a tagfüggvények működnek. Esetenként az ilyen környezet létrehozása valamilyen erőforrás lefoglalásával jár, amit a használat után fel kell szabadítani. Következésképpen némelyik osztálynak szüksége van egy olyan függvényre, amely biztosan meghívódik, amikor egy objektum megsemmisül, hasonlóan ahhoz, ahogy a konstruktor meghívására is biztosan sor kerül, amikor egy objektum létrejön: ezek a destruktor (megsemmisítõ, destructor) függvények. Feladatuk általában a rendbetétel és az erőforrások felszabadítása. A destruktorok automatikusan meghívódnak, amikor egy automatikus változót tartalmazó blokk lefut, egy dinamikusan létrehozott objektumot törölnek és így tovább. Nagyon különleges esetben van csak szükség arra, hogy a programozó kifejezetten meghívja a destruktort. A destruktor legjellemzőbb feladata, hogy felszabadítsa a konstruktorban lefoglalt memóriaterületet. (jelölés: ~ClassNeve())
•**Alapértelmezett konstruktor**: az a konstruktor, amelyiket paraméter nélkül hívhatjuk meg. Ha a programozó megadott alapértelmezett konstruktort, akkor a fordítóprogram azt fogja használni, máskülönben szükség esetén megpróbál létrehozni egyet
•**member initializer** (tag-kezdőérték lista): kötelező olyan tagokra, amelyeknek speciális a kezdőérték-adása, konstans, referencia; fontos a sorrend!

pl.: X::X(int a, int b):adattag1(a),adattag2(b){ } //az adattag1 értéke a, az adattag2 értéke b lesz
•ha egy tagfüggvényt **friend**ként határozunk meg, a függvény hozzáférhet az osztály deklarációjának privát részeihez. leggyakrabban: olyan operátor, amelynek első operandusa nem általunk írt osztálybeli
•**Származtatott osztályok**: A Bicikli a Járműből származik, és fordítva a Jármű a Bicikli **bázisosztálya**. A származtatott osztály tulajdonságokat örököl a bázisosztálytól. A származtatott osztályok tagfüggvényei ugyanúgy elérhetik a bázisosztály **public** és **protected** tagjait, mintha maguk vezették volna be azokat. A származtatott osztály azonban nem éri el a bázisosztály private tagjait. **A konstruktorok nem öröklődnek!** A származtatott osztály maga is lehet bázisosztály.
•A **virtuális függvények** segítségével a programozó olyan függvényeket deklarálhat a bázisosztályban, amelyeket a származtatott osztályok felülbírálhatnak. Egy virtuális függvény akkor szolgálhat felületként a származtatott osztályokban definiált függvényekhez, ha azoknak ugyanolyan típusú paramétereik vannak, mint a bázisosztálybelinek, és a visszatérési érték is csak nagyon csekély mértékben különbözik. A virtuális tagfüggvényeket néha metódusoknak is hívják. A virtuális függvényt mindig definiálnunk kell abban az osztályban, amelyben először deklaráltuk, hacsak nem tisztán virtuális függvényként adtuk meg. Virtuális függvényt akkor is használhatunk, ha osztályából nem is származtatunk további osztályt; ha pedig származtatunk, annak a függvényből nem kell feltétlenül saját változat. **Tisztán virtuális függvény**: nincsen definíciója az adott osztályban (pl.: virtual void fv()const = 0)
•**Absztrakt osztályok**: van legalább egy tisztán virtuális tagfüggvénye. Nem példányosítható, csak bázisosztályként használható!

SABLONOK

• A template<class C> előtag azt jelenti, hogy egy sablon deklarációja következik és abban a C típusparamétert fogjuk használni. Bevezetése után a C-t ugyanúgy használhatjuk, mint bármely más típusnevet. A C hatóköre a template<class C> előtaggal bevezetett deklaráció végéig terjed. (class és typename is használható)
• A sablonoknak lehetnek típust meghatározó, közönséges típusú (pl. int), és sablon típusú paramétereik. A sablon paramétere lehet konstans kifejezés, külső szerkesztésű objektum vagy függvény címe, illetve egy tagra hivatkozó, túl nem terhelt mutató. **Karakterlánc literált nem használhatunk sablonparaméterként.**
•példányosítás: implicit (típuskikövetkeztetéssel) pl. fv(x,y); explicit (típusválasztással) pl. fv<int>(x,y)

STL

•szekvenciális: std::list, std::vector, std::deque
•asszociatív: std::set, std::multiset, std::map, std::multimap
•adapter: std::stack, std::queue, std::priority_queue
szekvenciák: változó méretű konténerek, melyek elemei (szigorúan) meghatározott lineáris sorrendben követik egymást. Támogatják az elemek beszúrását és törlését
asszociatív tárolók: változó méretű konténerek, amelyek elemek (értékek) hatékony visszakeresését támogatják kulcsok alapján. Támogatott az elemek beszúrása, törlése, de a szekvenciáktól különböznek abban, hogy az elemeket nem meghatározott helyekre, pozíciókra szúrhatjuk be.

EGYÉB

•Állomány őrszemek (megszűnteti a többszörös include-okat; fontos a VMI_H egyedi legyen, különben nem működik)

```
// vmi.h:
#ifndef VMI_H
#define VMI_H
…
#endif // VMI_H
```

•tárolási-mód-meghatározás:
auto
register
static
extern
mutable
•függvény-meghatározás:
inline
virtual
explicit