Konstansok és konstans kifejezések. Heap kezelése. Pointerek és referenciák. Hibakezelés.

C++ programozás – 2. óra

Széchenyi István Egyetem
©Csapó Ádám

`http://dropbox.com/...`

2017

Konstansok és konstans kifejezések

- C++ két const-hoz kapcsolódó jelentést társít:
  - **const**: "*megígérem, hogy ezt az értéket nem változtatom meg*"
    - interfészek meghatározásánál nem kell attól tartani, hogy a fv-eknek átadott adat módosul
    - főleg pointereknél vagy referenciánál van értelme, mivel minden más érték szerint kerül átadásra
  - **constexpr**: "*értékeld ki fordításkor*"
    - konstansok megadásakor hasznos, illetve amikor olyan memória-területre akarunk írni, ami read-only ami (nem valószínű hogy corrupted lesz), illetve lehetnek teljesítménybeli előnyei is

```
const int x; //nem OK, konstanst muszaj inicializalni!
const int dmv = 17; //dmv nevvel ellatott konstans
int var = 17; //var nem konstans
constexpr double max1 = 1.4*square(dmv); //OK akkor ha square(dmv) fv is constexpr
constexpr double max2 = 1.4*square(var); //hiba: var nem constexpr, ezert ezt sem lehet kiertekelni
const double max3 = 1.4*square(var); //OK, futasidoben kiertekeljuk ha var erteke addigra ismert
double sum(const vector<double>&); //sum fv. nem modositja argumentumat
const double s1 = sum(v); //OK, futasidoben kiertekeljuk ha v erteke addigra ismert
constexpr double s2 = sum(v); //hiba: sum(v) nem konstans kifejezes
```

Konstansok és konstans kifejezések - II.

- constexpr megadásakor csak olyan fv használható, amely maga is constexpr.
- constexpr függvény nagyon egyszerű lehet csak, egyetlen return paranccsal (különben nehéz lenne fordításidőben kiértékelni). Példa:

```
constexpr double square(double x){return x*x;}
```

- constexpr fv meghívható nem konstans paraméterrel is, de olyankor az eredményt nem használhatjuk fel constexpr-ben. Magyarán a constexpr fv egyébiránt olyan, mint bármilyen függvény
- constexpr előnyei:
  - Névvel illettett konstansok könnyebben érthetőek és fenntarthatóak, mintha egy számot vagy értéket írnánk be több helyre
  - Ráadásul constexpr lokálisan scope-olható, nem úgy mint a *#define*

Konstansok és konstans kifejezések III.

- constexpr előnyei (folyt.):
  - A C++ amúgy is constexpr-et vár tömbök méretéhez, case címkékhez és template-ek érték-argumentumaihoz, ezért nem árt ha tudjuk hogy explicit módon is használható
  - Beágyazott rendszerekben hasznos, ha az adat csak olvasható memóriába kerül (rendszerösszeomlás után is megmarad, és kisebb energia-felhasználás is jellemzi, mint a dinamikus memóriát)
  - Többszálas rendszerben biztos hogy nincs versenyhelyzet ("*data race*"), ha fordításkor ismert az érték
  - Fordításidőben történő kiértékelés csak egyszer szükséges, míg futásidőben lehet h ugyanazt az értéket x milliószor ki kellene
- constexpr nagyon rugalmas egyébként:
  - lehet egész, lebegőpontos, vagy enum típusú is
  - bármilyen állapotmentes operátor alkalmazható rá (= és ++ pl. nem)
  - a ? : operátorral szelekció is végezhető (függetlenül attól hogy constexpr függvényben csak egy return statement lehet)

Konstansok és konstans kifejezések IV.

- Példa:

```
constexpr int isqrt_helper(int sq, int d, int a)
{
  return sq <= a ? isqrt_helper(sq+d, d+2, a) : d;
}

constexpr int isqrt(int x)
{
  return isqrt_helper(1, 3, x)/2 - 1;
}

constexpr int s1 = isqrt(9); //s1 erteke 3 lesz
constexpr int s2 = isqrt(1234);
```

- A függvény az egészre lefele kerekített négyzetgyököt adja vissza.
- Nem lényeges, de némi magyarázat hasznos:
  - Az sq értékek négyzetszámok
  - A d értékek pedig ezen négyzetszámok gyökei szorozva 2-vel plusz 1

Konstans mutatók

- Visszatérve const-ra, muszáj inicializálni (pl. simán “const int x;” nem ok)
- Mutatók esetében: nem mindegy, hogy egy mutató konstans (más címre később már nem mutathat), vagy az adat, amire mutat, konstans!

```
char *const cp; //konstans mutato, ami karakterre mutat (constant pointer)
char const* pc; //mutato, ami konstans karakterre mutat (pointer to const)
const char* pc2; //mutato, ami konstans karakterre mutat (pointer to const)
```

- Sokan használják a hátulról olvasás módszerét:
  - “cp egy konstans pointer karakterre”
  - “pc egy pointer konstans karakterre”
  - “pc2 egy pointer karakterre, amely konstans” (kétértelmű, előző jobb)

Konstans mutatók II.

- Konstansra mutató pointernek adható olyan cím, amely nem konstans változót tartalmaz
  - ebből baj nem származhat, max annyi lesz hogy a pointeren kersztül nem módosíthatjuk a változót
- Konstans változó címét azonban nem adhatjuk nem konstansra mutató pointernek, mert ezáltal a mutatott változó értéke módosítható lenne (dereferencia operátor útján!)

```
int a = 1;
const int c = 2;
const int* p1 = &c; //OK
const int* p2 = &a; //OK
int* p3 = &c; //hiba: int* valtozo inicializalasa const int*-el
*p3 = 7; //ez gond lenne, pont ezert hiba az elozo sor
```

Heap kezelese: new és delete

- Minden névvel ellátott objektum a scope-jának megfelelő élettartamú
- Sokszor hasznos azonban, ha scope-tól független élettartalmú objektumot hozunk létre
  - Pl. fv visszatérése után használható legyen, még ha a fv-en belül hoztuk is létre
- A **new** operátor éppen erre való. A memória-területet felszabadítani a **delete** operátorral lehet.
  - A new-val létrehozott objektumok a halomban (heap, avagy free store) helyezkednek el
  - A new olyan típusú pointert ad vissza, mint amilyen típusra az operátort meghívtuk! Működik alaptípusokra, osztályokra és tömbökre is. Pl:

Heap kezelese II: new és delete II.

```
int* pi = new int;
//...
delete pi;

vector<int>* pvi = new vector<int>(n);
//...
delete pvi;

char* sp = new char[5];
//...
delete[] sp;
```

- Ahhoz, hogy a delete működjön, a futtatási környezetnek ismernie kell a new-val lefoglalt terület méretét. Ezért az így létrehozott változók, objektumok kicsit nagyobbak, mint a stack-en vagy adatszegmensben létrehozottak. Legtöbbször nem jelentős overhead, de sok kis objektum esetén korlátot szabhat.
- Megvalósításuk a <new> header alatt található.

Heap kezelese III: new és delete III.

- Ha a new operátor nem tud több területet foglalni, bad_alloc típusú kivételt dob:

```
void f()
{
  vector<char* >v;
  try{
    for(;;){
      char* p = new char[10000];
      v.push_back(p); //ugyelunk arra hogy maradjon referenciank a mem.teruletre
      p[0] = 'x';
    }
  }
  catch(bad_alloc){
    std::cerr << ``Memory exhausted!'' << std::endl;
  }
}
```

- Nem árt vigyázni: ha van virtuális memória is, akkor a merevlemezre kezd el írogatni a program, és csak akkor dob kivételt ha az is betelik

Heap kezelése IV: mi a gond a new és delete operátorokkal?

- A halomban tárolt pointerekkel rengeteg gond van: nehéz a new operátorokkal lefoglalt memóriát észben tartani és felszabadítani.
- Több hibalehetőség is van:
  - new használatát követően a delete-ről elfelejtkezünk (memória-szivárgás). Vagy ha new-t követően kivételt dob a kód?
  - korai delete: meghívjuk a delete-et, de még létezik arra a memória-rekeszre mutató másik pointer, amit később használunk. Ez könnyen baj lehet, mert a futtatási környezet használhatja azt a mem.területet másra!
  - kétszeres delete: rossz ha nem idempotens a destruktor

```
int* p1 = new int{99}; int* p2 = p1; //ajjaj...
delete p1; //p2 ezek utan nem mutat konzisztens allapotra!
p1 = nullptr; //hamis biztonsagerzetet ad
char* p3 = new char{'x'}; //itt akar felul is irhattuk az eredeti 99-et!
*p2 = 999; //itt lehet hogy ezert pont a karaktert irjuk felul
std::cout << *p3 << std::endl; //nem biztos, hogy 'x'-et kapunk
```

Heap kezelése V: mi a gond a new és delete operátorokkal? II.

```
void sloppy()
{
  int* p = new int[1000];
  //...
  delete[] p;

  //... kicsit varunk

  delete[] p; //de sloppy() mar nem is birtokolja *p-t!
}
```

- Megoldási lehetőségek:
  - Használjunk ún. manage-elt objektumot, amely handle-t ad a területre és garantáltan törli a scope megszűnésekor. Példák:
    - string, vector és egyéb STL konténerek
    - unique_ptr, shared_ptr (C++11 óta)
    - külső kvázi-sztenderd könyvtárak: boost shared ptr
  - Használjunk inkább referenciát.

Heap kezelése VI: RAII technika

- Az 1. módszert RAII technikának is nevezik (resource allocation is initialization). Sokak szerint ez rossz elnevezés, inkább Destruction is Resource Relinquishment jobb lenne. De a lényeg:

```
class someResource
{//belso reprezentaciok, ptr-ekkel, stb.
public:
  someResource(){//...eroforrasok lefoglalasa}
  ~someResource(){//... eroforrasok felszabaditasa}
};
```

- Az STL vector osztály ilyen! *push_back()* hívások halomra mennek:

```
void f(const string& s)
{
  vector<char> v;
  for(auto c: s)
    v.push_back(c);
}
```

Referenciák

- További lehetséges megoldás heap és pointerekkel kapcs. gondokra.
- Előbb: néhány további gond pointerekkel:
  - Eltérő szintaxist használunk eltérő helyzetekben, pl. *p-t, ha p objektum, de p->m-et, ha m egy p által mutatott objektum tagváltozója
  - Egy pointer különböző időpontokban más objektumokra mutathat
  - Pointer értéke bármikor lehet nullptr, amire ügyelni kell (ld. előző pontot)
- Referenciák olyanok mint a pointerek, csak:
  - Referencia ugyanolyan szintaxissal hivatkozható, mint maga az obj.
  - Referencia mindig arra az objektumra mutat, amelyikkel inicializáltuk
  - Nem létezik “null reference”, ezért mindig feltételezhetjük, hogy tényleg objektumra mutat

Referenciák II

- T típusú objektumra mutató referencia: T&
- Ha függvénynek referenciát adunk át, a fv módosíthatja annak a változónak az értékét, amelyre a referencia tul.képpen egy alias (tehát úgy is gondolhatunk rá, mint egy const pointerre, amely mindig "dereferenciálódik")
- Az lvalue/rvalue és const/nem const megkülönböztetésére 3 referencia-típus létezik:
  - **lvalue referencia**: megváltoztatandó objektumokra hivatkozik
  - **const referencia**: konstans objektumokra hivatkozik
  - **rvalue referencia**: olyan objektumokra hivatkozik, melyek értékét a használat után nem akarjuk felhasználni (pl. temp változók)

Referenciák III

- T típusú objektumra mutató lvalue referencia: T&

```
int var = 1;
int& r{var}; //lehetne = var is, de a lenyeg: r es var mostmar u.arra az intre hivatkoznak
int x = r; //x erteke mostantol 1
r = 2; //var erteke mostantol 2
```

- A referenciának muszáj valamire hivatkoznia, tehát nem elég deklarálni, inicializálni is kell

```
int var = 1;
int& r1{var};
int& r2; //hiba: hianyzik az inicializalas!
extern int& r3; //OK: r3 mashol mar inicializalva volt
```

Referenciák IV

- Mint mondtuk, referencia maga az objektum (nincs rá külön operátor)

```
int var = 0;
int& rr{var};
++rr; //var-t inkrementaljuk 1-re
int* pp = &rr; //pp var-ra mutat
```

- T& referencia inicializálása csak lvalue-n keresztül tehető meg (olyan objektum, melynek a címe lekérdezhető):

```
int& r1 = 1; //hiba, mert 1 nem lvalue
int i = 1;
int& r2 = i; //igy mar mas
```

Referenciák V

- Konstans T típusú referencia inicializálásakor nem kötelező sem lvalue-t, sem pedig (feltétlenül) T típust megadnunk (rvalue használható mint temporary object, és implicit konverzió is létezik, pl. int-ről double-ra):

```
double& dr = 1; //nem OK
const double& cdr {1}; //OK
```

- De miért ne lehetne nem const referenciákhoz is temp változót létrehozni?
  - Mert ha a referencia értékét módosítjuk, a temp változóét is módosítanánk, ami nagyon sok extra könyvelést jelentene a futtatási környezetnek
  - Ráadásul nincs is a C++ szabványban kikötve hogy hogyan kellene a temp objektumokat tárolni!

Referenciák VI: Mire jók a referenciák?

- Paraméterként használva: függvény megváltozathatja az értékét. Csak akkor használjuk, ha a függvény neve kellően beszédes.
- Például itt a next() sokkal érthetőbb, mint az increment(), mivel az utóbbi semmilyen értéket nem ad vissza

```
void increment(int& aa){
  ++aa;
}

int next(int p){
  return p+1;
}

void g(){
  int x = 1;
  increment(x); //x = 2;
  x = next(x); //x = 3;
}
```

Referenciák VII: Mire jók a referenciák? II

- Paraméterként használva, folyt.: ha nagy adatstruktúráról van szó, olcsóbb lehet referenciát átadni, mint érték szerint a stack tetejére másoltatni!
- Visszatérési értékként is jók abból a célból, hogy a fv lvalue-ként és rvalue-ként is használható legyen. Pl. Map osztály esetén:

```
template<class K, class V>
class Map{ //egyszeru Map osztaly
public:
  V& operator[](const K& k); //visszaadja a k kulcshoz tartozo erteket
  //ekkor is: mymap[5] = xyz;
  //meg ekkor is: xzy = mymap[5];

  pair<K,V>* begin(){return &elem[0];}
  pair<K,V>* end(){return &elem[0]+elem.size();}

private:
  vector<pair<K,V> > elem; //maguk a kulcs,ertek parok
};
```

Referenciák VIII: Mire jók a referenciák? III

- Egy lehetséges megvalósítás (a sztenderd megvalósítás ún. red-black tree adatstruktúrát használ, itt a szemléltetés kedvéért viszont szekvenciális keresést használunk)

```
template<class K, class V>
V& Map<K,V>::operator[](const K& k)
{
  for(auto& x: elem){
    if(x.first == k){
      return x.second;
    }
  }
  elem.push_back({k,V{}}); //ha nem talaljuk a kulcsot, adjunk egy uj pair-t elem vegehez
  return elem.back().second; //adjuk vissz az uj elem default erteket
}
```

- k argumentum referencia, mert nem tudjuk milyen típus (költséges lehet másolni). A kimenet is ezért is V& referencia

Referenciák IX: Mire jók a referenciák? IV

- k ugyanakkor const, mert értékét nem módosítjuk, és mert lehet hogy nem lvalue értéket szeretnénk átadni az operátornak
- A kimenet viszont nem const, mert lehet hogy a felhasználó módosítaná a kapott értéket
- Például használhatjuk így is:

```
int main() //szamoljuk meg hanyszor szerepel egy-egy szo a bemenetben
{
  Map<string, int> buf;

  for(string s; std::cin >> s; )
  {
    ++buf[s];
  }

  for(const auto& x : buf){
    std::cout << x.first << ": " << x.second << std::endl;
  }
}
```

Referenciák X.: rvalue referenciák

- Végül: rvalue referenciák.
  - Non-const lvalue: írható referencia
  - const lvalue: hivatkozható, de nem írható
  - rvalue: temp objektum, ami módosítható
- De miért fontos ez? Azért, mert ha tudjuk, hogy a hivatkozott adatot többet nem használjuk, akkor az adat “elmozdítható” (és megspórolható a másolása)

Referenciák XI: rvalue referenciák II

- rvalue referencia csak rvalue alapján hozható létre, jelölése &&

```
string var{"Cambridge"};
string f();

string& r1{var}; //lvalue referencia, r1 var-ra hivatkozik
string& r2{f()}; //lvalue referencia, de hiba: f() rvalue-nak szamit!
string& r3{"Princeton"}; //lvalue referencia, de hiba: nem hasznalhatunk temp objektumot!

string&& rr1{f()}; //rvalue referencia, es OK: rr1 erteke egy temp objektum
//mivel az objektum temp, ezert mashol nem szamithatunk ra es ha masolni kell, inkabb
//csak "atcimkezi" a futtatasi kornyezet
string&& rr2{var}; //rvalue referencia, de hiba: lvalue erteket adtunk neki
string&& rr3{"Oxford"}; //rendben, rvalue referencia egy temp objektumra

const string& cr1{"Harvard"}; //OK, csinalunk egy temp objektumot es hozza tartsitjuk cr1-hez
```

Referenciák XII: rvalue referenciák III

- const rvalue referencia nincs, mert leggyakrabban egy temp érték átírására szeretnénk csak használni (mozgatás)
- Fontos: const lvalue és rvalue referenciák is mutathatnak rvalue-ra. De a felhasználásuk merőben különbözhet!
  - rvalue referencia: destruktív olvasás (másolás helyett mozgatás)
  - const lvalue referencia: argumentum módosítását elkerülendő

```
std::string f(std::string&& s)
{
  if(s.size()){
    s[0] = toupper(s[0]);
  }
  return s; //sem a belso, sem az eredeti s-t(!) mar soha az eletben nem fogjuk hasznalni
  //ezert ha a futtatasi kornyezet eleg okos, nem kell mas valtozoba masolnia a kimeneti erteket,
        eleg csak atcimkeznie!
}

//igy a fv meghivhato rvalue-val inicializalt stringgel (sot, csak azzal):
std::cout << f("muuuukodj!");
```

Referenciák XIII: rvalue referenciák IV

- Néha olyan is van, hogy a programozó tudja, hogy egy objektumot többet nem fog használni, de a fordító nem. Pl.

```
template<class T>
void swap(T& a, T&b) //old-style swap
{
  T tmp{a}; //ezutan ket kopia van a-bol
  a = b; //ezutan ket kopia van b-bol es egy a-bol
  b = tmp; //ezutan ket kopia van tmp-bol (ami valojaban a)
}

//ezert ehelyett

template<class T>
void swap(T& a, T&b)
{
  T tmp{static_cast<T&&>(a)}; //a ertekere mashol mar nem lesz szuksegunk ezert erteke ``ellophato''
  a = static_cast<T&&>(b); //b ertekere mashol nem lesz szuksegunk ezert erteke ``ellophato''
  b = static_cast<T&&>(tmp); //tmp erteke mashol nem kell, ezert mozgatas helyett direktben tmp
        memoriarekeszere hivatkozunk ezutan a nevvel
}
```

Referenciák XIV: rvalue referenciák V

- Persze könnyű félregépelni a static_cast<T&&>-t. Ezért a standard könyvtárnak van egy move() függvénye, ami ugyanezt csinálja:

```
template<class T>
void swap(T& a, T& b)
{
  T tmp{std::move(a)};
  a = std::move(b);
  b = std::move(tmp);
}

//vegul, a megoldas csak majdnem tokeletes, mert csak lvalue parameterekkel mukodik, pl.
void f(vector<int>& v)
{
  swap(v, vector<int>{1,2,3}); //ez nem mukodik mert a 2. argumentum nem lvalue
  //ezert vagy tobb swap fv-t definialunk, vagy egyeb trukkoket alkalmazunk ami most messzire
        vezetne
}
```

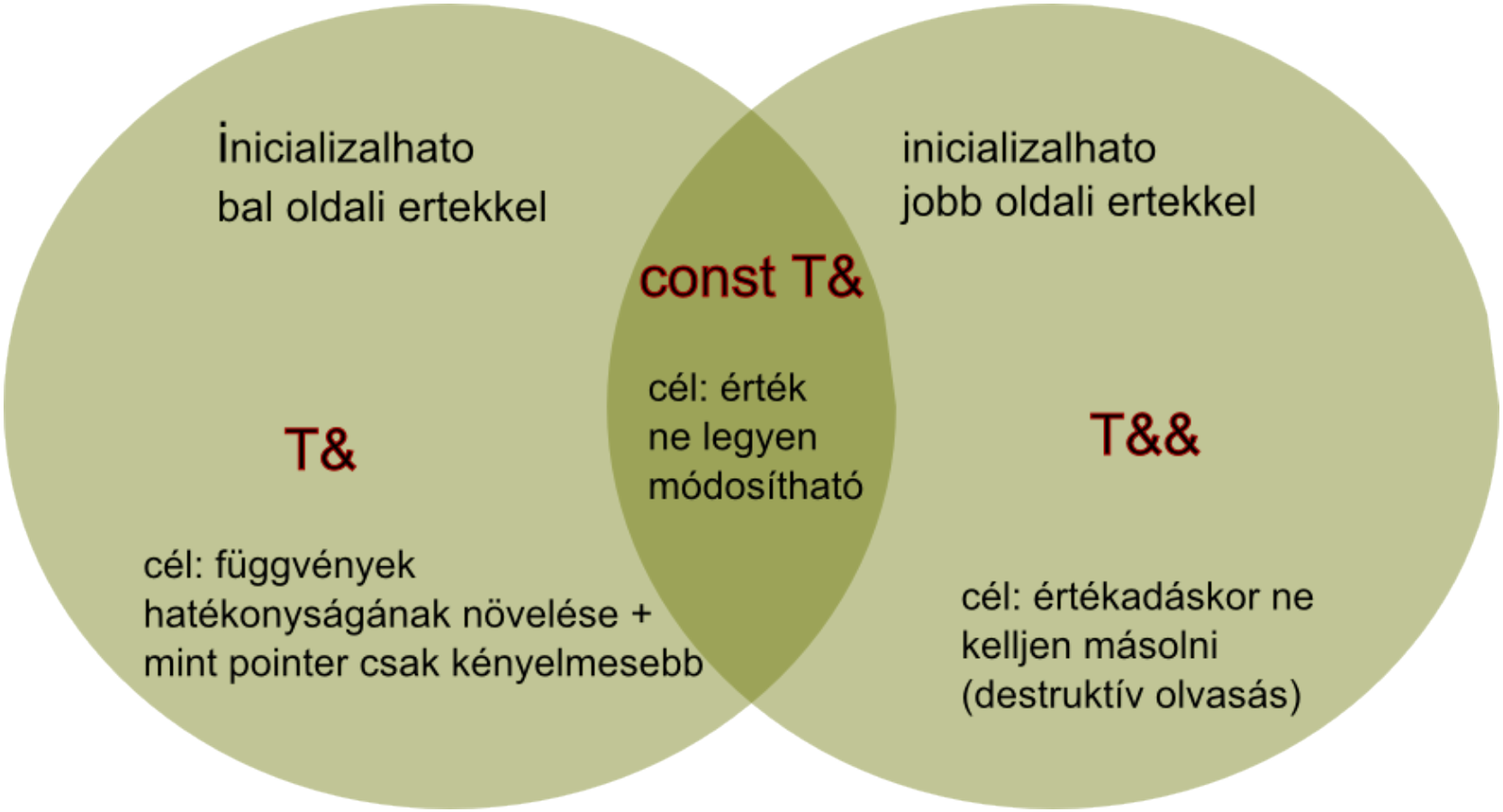
Referenciák XV

- Végül: referencia referenciája is ugyanolyan referencia (tehát nem a referenciára mutató valami, hanem ugyanarra az objektumra hivatkozik). De milyen típusú referencia lesz?
  - Mindig lvalue referencia győz ("reference collapse")! Tehát:

```
int && & r = i; // nem OK!!
```

- Néhány további ökölszabály referenciák használatára:
  - Referencia-paraméter fv-ben módosítható → nehéz olvashatóság!
  - Ezért: ha nem módosítjuk, mindig használjunk const-ot. Ilyenkor tudjuk, hogy hatékonyság miatt adtuk át referenciaként
  - Ha ezt betartjuk, akkor viszont const hiánya esetén tudjuk: a fv módosítani fogja a paramétert!

Referenciák: Összefoglaló ábra



Pointerek és referenciák: mikor melyiket használjuk?

- Erre a kérdésre létezik néhány ökölszabály illetve preferencia.
- Ha egy függvényben át szeretnénk irányítani egy hivatkozást másik objektumra, használjunk pointert!
  - Ilyenkor használhatóak a ++, – stb. operátorok

```
void fp(char* p)
{
  while(*p)
  {
    std::cout << *p++;
  }
}

void fr(char& r)
{
  while(r)
  {
    std::cout << ++r; //jaj! a hivatkozott karaktert noveljuk, nem a referenciat!
    //majdhogynem vegtelen ciklus
  }
}
```

Pointerek és referenciák: mikor melyiket használjuk? II

- Ellenben, ha garantáltan mindig ugyanarra az objektumra akarunk hivatkozni, használjunk referenciát!
  - Hiszen a referencia inicializálást követően nem változtatható!

```
template<class T>
class Proxy //Proxy az inicializalt objektumra hivatkozik
{
  T& m;

public:
  Proxy(T& mm) : m(mm){}
  //...
};

template<class T>
class Handle //Handle az aktualis objektumra hivatkozik
{
  T* m;

public:
  Proxy(T* mm): m(mm){}
  void rebind(T* mm){m = mm;}
};
```

Pointerek és referenciák: mikor melyiket használjuk? III

- Ha operátor-felültöltéssel szeretnénk műveleteket végezni valamin, ami objektumra hivatkozik, használjunk referenciát (a pointer beépített típus, nem használható op.-felültöltésre)!

```
Matrix operator+(const Matrix&, const Matrix&); //OK
Matrix operator-(const Matrix*, const Matrix*); //hiba: nincs UDT argumentum!

Matrix y,z;
//...
Matrix x = y+z; //OK
Matrix x2 = &y - &z; //hiba, es meg ronda is...
```

Pointerek és referenciák: mikor melyiket használjuk? IV

- Ha olyan kollekciót akarunk, ami objektumokra hivatkozó dolgokból áll, muszáj pointert használni

```
int x, y;
string& a1[] = {x, y}; //hiba: referenciak tombje nem hozhato letre
string* a2[] = {&x, &y}; //OK
vector<string&> s1 = {x, y}; //hiba: referenciak vektora nem hozhato letre
vector<string*> s2 = {&x, &y}; //OK
```

Pointerek és referenciák: mikor melyiket használjuk? V

- Fentieken túlmenően tényleg ízlés dolga. Ami számíthat, hogy vannak apró különbségek, pl. pointerek rendelkeznek "nulla" értékkel (*nullptr*), referenciák viszont nem. Pl.

```
void fp(X* p)
{
  if(p==nullptr)
  {
    //nincs erteke
  }else
  {
    //hasznalhatjuk *p-t
  }
}

void fr(X& r) //altalanos stilus
{
  //feltetelezzuk, hogy r letezik es hasznalhatjuk
}
```

Pointerek és referenciák: mikor melyiket használjuk? VI

- Pointer mutathat pointerre, referencia mutathat referenciára, referencia mutathat pointerre, DE: pointer nem mutathat referenciára
  - hiszen a referencia nem típus (csak egy név ami valamire utal) és ezért nincs is címe
  - ráadásul lehetnek pl. jobb oldali referenciák is!
- Ezek egyébként hasznos dolgok.
- Pl. ha csak pointert adunk át egy függvénynek, a pointert is csak érték szerint adjuk át... ezért ha módosítjuk a pointer címét, annak nem lesz kihatása a hívó környezetre (csak annak lesz, ha a mutatott értéket módosítjuk)
  - Ezért szoktunk pointerre pointert, vagy pointerre referenciát átadni.

Pointer vagy referencia mutat pointerre

- Sima pointer nem jó:

```
//globalis valtozo: ezt keruljuk, most csak pelda erdekeben
int g_One = 1;

//fv prototipus
void func(int* pInt);

int main()
{
  int nvar = 2;
  int* pvar = &nvar;
  func(pvar);
  std::cout << *pvar << std::endl; //meg mindig 2

  return 0;
}

void func(int* pInt)
{
  pInt = &g_One;
}
```

Pointer vagy referencia mutat pointerre II

- Ezért inkább átadhatunk pointerre pointert:

```
//fv prototipus
void func(int* pInt);

int main()
{
  int nvar = 2;
  int* pvar = &nvar;
  func(&pvar);
  //...
  return 0;
}

void func(int** ppInt)
{
  *ppInt = &g_One; //ppInt mutato erteke most mar g_One-ra mutat

  *ppInt = new int; //igeny szerint meg memoria is allokalhato

  **ppInt = 3; //de mint korabban a mutatott mutato altal mutatott ertek is felulirhato
}
```

Pointer vagy referencia mutat pointerre II

- De ugyanígy használhatunk pointerre hivatkozó referenciát is (ízlés kérdése)

```
//fv prototipus
void func(int* pInt);

int main()
{
  int nvar = 2;
  int* pvar = &nvar;
  func(pvar);
  //...
  return 0;
}

void func(int*& rpInt)
{
  rpInt = &g_One; //rpInt altal hivatkozott mutato erteke most mar g_One-ra mutat

  rpInt = new int; //igeny szerint meg memoria is allokalhato

  *rpInt = 3; //de mint korabban a referalt mutato altal mutatott ertek is felulirhato
}
```

Hibakezelés

- Szoftverhibákat futás közben is szeretnénk észlelni és "felállni" belőlük.
- Alapból segíthetnek persze a statikus ellenőrzések, de mi a helyzet a dinamikus problémákkal? *dynamic_cast* is ugye *nullptr*-t ad vissza hiba esetén, és ha ahhoz hozzá szeretnénk férni...(!)
- További gond: egy "könyvtár" írója sokszor nem tudhatja, hogy mire akarjuk használni... a hiba forrása a felhasználásban rejlik.
  - Olyan is van, hogy tudja, de nem tudja könnyen detektálni a hibát.
- Ötlet: az ilyenkor "dobott" hibát elkapjuk és csinálunk valamit hogy a futás ne álljon le.

Hibakezelés II

```
void taskmaster()
{
  try
  {
    auto result = do_task();
    //eredmeny felhasznalasa
  }catch(Some_error){
    //hiba: problema kezelese
  }
}

int do_task()
{
  //...
  if(/*ha sikerult elvegezni a feladatot*/)
  {
    return result;
  }else{
    throw Some_error();
  }
}
```

- Ha *do_task*() más hibatípust is dob, *taskmaster*() nem fogja elkapni, ézért az azt hívó fv-nek kell valamit csinálnia.

Hibakezelés III

- Fontos persze az is, hogy az exception ne csak úgy “dobódjon”... alapesetben a program nem áll le hibaüzenettel, hanem folytatódik: ezért ügyelni kell arra, hogy elkapásakor az erőforrások ne szivárogjanak, stb.
- Jó esetben a kódunk által dobott exception típusa felhasználó által definiált.
  - Így világos lesz a jelentése, és nem keverjük össze más könyvtárak hiba-típusaival. Alapesetben nagyon egyszerű is lehet:

```
struct Range_error{};

void f(int n)
{
  if (n < 0 || max <= n) throw Range_error{};
  //...
}
```

Hibakezelés IV

- Exception lehet bármilyen típusú objektum, ami másolható! Az osztályok tagváltozói a konkrét hiba részleteit írják le
- Saját UDT (user-defined type) alkalmazása azért célszerű, mert:
  - Így nem keveredik össze több könyvtár hibaüzenete (képzeljük el, hogy megkapjuk hogy a hiba "17")
  - Ha API-t írunk, hasznos világossá tenni hogy olyan hiba keletkezett amivel mi számoltunk, vagyis azért van hiba mert az API-t nem rendeltetésszerűen használták, nem azért mert mi voltunk bénák
- Ha mégis túl melós saját exception-öket definiálni, a sztenderd könyvtár többféle exception osztályt definiál, hierarchikusan.
  - Régebben léteztek más alternatívák is (pl. globális hibaváltozó mint C-könyvtárakban *errno*, de akkor azt mindig ellenőrizni kellett).
- C++-ban nincs olyan, hogy finally blokk: inkább használjuk a RAII idiómát az erőforrások elengedésére!

Assert

- Végül megemlítendő az *assert* makró, amely feltételek és invariánsok általános célú ellenőrzését teszi lehetővé.
- Egy *assertion* olyan logikai kifejezés, amiről feltételezzük, hogy igaz
- $<$ *cassert* $>$-ben a sztenderd könyvtár definiálja az assert(A) makrót, amely csak akkor engedi tovább a futást, ha A teljesül, ellenkező esetben a program kilép
- Fordításidejű ellenőrzésre való a *static_assert*(*A*, *message*) hívás (C++11 óta). Ha A nem teljesül fordításkor, a message üzenetet kapjuk vissza.
- Ezek meglehetősen fapados megoldások, ennél jobb lehet saját Assert namespace-t létrehozni kényelmi funkciókkal (pl. be lehessen állítani biztonsági fokozatot, stb.)

Assert II

- Itt van egy példa, de erre majd visszatérünk ha a template-eket tanuljuk!

```
namespace Assert{
  enum class Mode{throw_, terminate_, ignore_};
  constexpr Mode current_mode = CURRENT_MODE;
  constexpr int current_level = CURRENT_LEVEL;
  constexpr int default_level = 1;

  constexpr bool level(int n)
  {
    return n <= current_level;
  }

  struct Error : runtime_error
  {
    Error(const string& p) : runtime_error(p) {}
  };
}
```

- *Assert* :: *dynamic* akkor működik, ha a hibajelzési szint megfelelő:

Assert II

```
namespace Assert{
  //...

  string compose(const char* file, int line, const string& message)
  {
    //osszedobjuk az uzenetet, a file nevevel es kodsor szamaval
    std::ostringstream os("(");
    os << file << "," << line << "):" << message;
    return os.str();
  }

  template<bool condition = level(default_level), class Except = Error>
  void dynamic(bool assertion, const string& message = "Assert::dynamic failed")
  {
    if(assertion){
      return;
    }
    if(current_mode==Assert_mode::throw_){
      throw Except{message};
    }
    if(current_mode==Assert_mode::terminate_){
      std::terminate();
    }
  }
}
```

Assert III

- ...és még néhány sor, aztán a felhasználás

```
namespace Assert
{
  //...
  template<>
  void dynamic<false, Error>(bool, const string&){} //semmit se teszunk

  void dynamic(bool b, const string& s) //alapertelmezett valasz
  {
    dynamic<true, Error>(b, s);
  }

  void dynamic(bool b){ //alapertelmezett uzenet
    dynamic<true, Error>(b);
  }
}

void f(int n)
{
  //n muszaj hogy 1 es max kozott legyen
  Assert::dynamic<Assert::level(2), Assert::Error>(
  (n <= 0 || max < n), Assert::compose(__FILE__, __LINE__, "range_problem")
  );
}
```

Feladatok

- Próbáljunk ki:
  - Egy constexpr-es példát (pl. 5-ös fólia)
  - Próbáljuk ki a konstans mutatókkal való trükközéseket (7. fólia) – tényleg hibát dob-e a fordító, stb.
  - Próbáljuk ki a new és delete operátorokat, túlcsordulást is (9., 10., 11. fóliák)
  - Próbáljuk ki a referenciákat (24. oldal) – vigyázat: inicializáló jelölés még lehet hogy nem működik
- Kezdjünk el felépíteni egy egész vektorokból, illetve rajtuk elvégezhető műveletekből álló osztályt!
  - Osztályt persze még nem tudunk, de amit igen:
  - Készíthetünk olyan fv-eket, melyek int[] tömböket adnak össze, vonnak ki, szoroznak össze, stb.
  - Cél: pointerek, referenciák, hibakezelés gyakorlása