Objektum orientált programozás C++ nyelven
2. Program struktúra

PEKÁRDY MILÁN – PANNON EGYETEM

PEKARDY@DCS.UNI-PANNON.HU

Kifejezések és vezérlési szerkezetek

- A program végrehajtása során pontosvesszővel lezárt utasítások futnak le a definiálásuk sorrendjében
- A különböző vezérlési szerkezetek lehetővé teszik, hogy bizonyos feltételek teljesülése esetén fussanak csak le egyes kódrészletek, ismételjük a kód egyes részeit, stb.
- Alapvető vezérlési szerkezetek:
  - if…else
  - while
  - do…while
  - for
  - ranged-based for
  - switch…case
  - break
  - continue
  - goto

Vezérlési szerkezetek (if…else)

- Az egyes kódrészletek csak a megadott feltételek teljesülése esetén hajtódnak végre
  - `if (condition) { statement(s) }`
- Az else és else if kulcsszavakkal több feltételt és végrehajtási ágat is megadhatunk

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

Vezérlési szerkezetek (while)

- Egy utasítás blokk addig ismétlődik, amíg a megadott feltétel teljesül
  - `while (expression) statement`
- Lehet, hogy egyszer sem hajtódik végre az utasítás blokk

```
// custom countdown using while
#include <iostream>
using namespace std;

int main ()
{
  int n = 10;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "liftoff!\n";
}
```

Vezérlési szerkezetek (do…while)

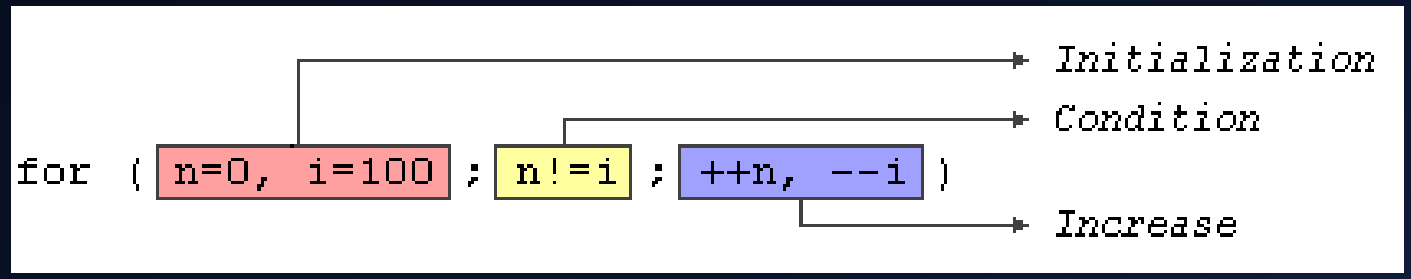
- Egy utasítás blokk addig ismétlődik, amíg a megadott feltétel teljesül
  - `do statement while (condition);`
- Az utasítás blokk legalább egyszer biztosan lefut

```
// echo machine
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
  do {
    cout << "Enter text: ";
    getline (cin,str);
    cout << "You entered: " << str << '\n';
  } while (str != "goodbye");
}
```

Vezérlési szerkezetek (for)

- Egy utasítás blokk addig ismétlődik, amíg a megadott feltétel teljesül
  - `for (initialization; condition; increase) statement;`
- A for ciklus működése:
  - initialization utasítások végrehajtódnak a ciklus kezdete előtt egyszer
  - condition részben definiált feltétel ellenőrzése: ha igaz, akkor folytatódik a ciklus, ha hamis akkor megáll
  - statement blokk utasításai végrehajtódnak
  - increase utasítások végrehajtódnak minden ciklus végén, majd a 2. ponttól ismétlődik



Vezérlési szerkezetek (for)

```
// countdown using a for loop
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

```
for ( n=0, i=100 ; n!=i ; ++n, --i )
{
   // whatever here...
}
```

Vezérlési szerkezetek (range-based for)

- Ez a fajta ciklus végig iterál a megadott tartomány elemein
  - `for ( declaration : range ) statement;`
- Minden iterációban a declaration részben megadott változó felveszi az aktuális elem értékét így azzal lehet műveleteket végezni a ciklus utasításaiban
- Meg van határozva, hogy milyen típusok kerülhetnek a range részbe (tömbök, tárolók, karakterláncok, saját típusok, stb.)

```
string str {"Hello!"};
for (char c : str)
{
    cout << "[" << c << "]";
}
```

Vezérlési szerkezetek (break)

- Egy ciklusból tudunk „kiugrani” a break kulcsszóval, még akár akkor is ha a ciklus feltétele igaz lenne
- Hasznos lehet ha egy végtelen ciklust kell befejezni bizonyos feltételek teljesülése esetén

```
// break loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
}
```

Vezérlési szerkezetek (continue)

- A continue kulcsszó segítségével kihagyhatjuk a ciklusmag kulcsszó után következő részét és új iterációval folytatódik a végrehajtás

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

Vezérlési szerkezetek (goto)

- A goto kulcsszó segítségével a program egy tetszőleges pontjára ugorhatunk
- A mai modern magasabb szintű programozásban kerülendő a használata
- Ha mégis szükséges, megfelelő körültekintéssel tegyük (nem veszi figyelembe az egymásba ágyazott blokkokat, nincs automatic stack unwinding, stb.)

```
int n=10;
mylabel:
cout << n << ", ";
n--;
if (n>0) goto mylabel;
cout << "liftoff!\n";
```

Vezérlési szerkezetek (switch-case)

- Egy kifejezés értékét hasonlítja konstans kifejezésekhez, és ha egyezést talál, akkor a megfelelő ág hajtódik végre, ha nincs egyezés, akkor megadhatunk egy alapértelmezett ágat is
- Egy switch-case blokk megfeleltethető egy if-else blokknak
- Általános szerkezet:

```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  .
  .
  .
  default:
     default-group-of-statements
}
```

Vezérlési szerkezetek (switch-case)

```
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }
```

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
}
```

Függvények

- Egy függvény utasítások csoportja, aminek nevet adunk majd a program egyéb pontjain a megadott névvel meghívhatjuk
- `type name ( parameter1, parameter2, ...) { statements }`
  - type: a függvény által visszaadott érték típusa
  - name: függvény neve, amivel meghívható
  - parameters: minden paramétert egy típus és egy azonosító definiál egy vesszővel elválasztott listában, a paraméterek segítségével adunk át adatokat a függvénynek a hívó kódból
  - statements: a függvény törzse, ami a függvény által végrehajtandó utasításokat tartalmazza

Függvények

- Függvény hívásakor a nevével hivatkozunk a függvényre
- A híváskor átadjuk a függvénynek a definiált típusoknak megfelelő paramétereket
- Ha van visszatérési érték akkor azt a szokásos módon hozzárendelhetjük egy megfelelő típusú változóhoz

```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```

Függvények

```
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return r;
}

int main ()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction (7,2) << '\n';
  cout << "The third result is " << subtraction (x,y) << '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
}
```

Függvények

- Olyan függvényt is megadhatunk, amelyiknek nincs visszatérési értéke, ilyenkor a void kulcsszót használjuk

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
}
```

Függvények

- A main függvénynek int típusú visszatérési értéke van, de ebben az esetben elhagyható a return kifejezés. Hiba nélküli végrehajtás esetén a fordító implicit feltételezi a return 0; kifejezést
- Minden más függvénynél, aminek void-tól eltérő visszatérési értéke van kötelező explicit return megadása
- main visszatérési értékei:

value	description
0	The program was successful
EXIT_SUCCESS	The program was successful (same as above). This value is defined in header <cstdlib>.
EXIT_FAILURE	The program failed. This value is defined in header <cstdlib>.

Függvények

- A függvények paramétereinek az átadása történhet érték szerint, illetve referencia szerint
- Alapesetben a paraméterek érték szerint adódnak át, azaz a függvény hívásakor az értékek átmásolódnak a függvény paramétereibe, így a függvény által végzett módosítások csak a függvénytörzsben látszódnak azon kívül változatlanok az értékek
- Bizonyos esetekben kívánatos lehet, hogy a függvény módosítsa a megkapott paraméter értékeket, ilyenkor referencia szerint adjuk át az értékeket
  - a referenciaként átadandó paramétereket az & karakterrel jelöljük

Függvények

```
// passing parameters by reference
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

Függvények

- Érték szerinti paraméter átadáskor a változók másolódnak. Ez az alapvető típusok esetén nem problémás viszont összetettebb, nagyobb méretű adatot reprezentáló típusok esetén erőforrás igényes lehet ezek másolása.
- Megoldás lehet a referenciaként való átadás, de amikor nem akarjuk, hogy a függvény módosítsa az értékeket, akkor ez nem megfelelő megközelítés.
- A fenti problémák kiküszöbölésére használhatunk konstans referenciákat, ami azt jelenti, hogy az értékek referenciaként adódnak át, de a függvény nem módosíthatja a tartalmukat, így megoldva a másolás és módosítás problémáit

```
string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

Függvények

- Egy függvény hívása és végrehajtása plusz munkával jár ahhoz képest mintha csak a main-be írtuk volna a függvény utasításait
- Rövid függvények esetén hatékonyabb lenne ha inkább beszúrnánk a függvénytörzset a hívás helyére így megspórolva a függvény hívást
- Az inline kulcsszó segítségével javasolhatjuk a fordítónak, hogy inkább fejtse ki a függvényt a hívások helyén így megmarad a függvények által nyújtott újrafelhasználhatóság, de elhagyjuk a függvényhívással járó többlet munkát
  - a fordító dönthet úgy, hogy inline nélkül is optimizálja a kódot

```
inline string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

Függvények

- Egy függvénynek lehetnek opcionális paraméterei is, ilyenkor a függvény hívásakor az opcionális paramétert nem kell megadni. Ahhoz, hogy ez működjön a függvény paraméter listájában alapértelmezett értéket kell megadni az utolsó paraméterhez

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  return 0;
}
```

Függvények

- Ahhoz, hogy a függvényeket használjuk deklarálnunk kell őket (a használat helye előtt)
- Az eddigi példákban a függvények a törzsükkel együtt a main előtt voltak deklarálva, így ott már hívhatóak voltak
- Amikor a függvényt egy másik helyen fejtjük ki, akkor a deklarálást a függvény prototípusának megadásával végezzük

```
// declaring functions prototypes
#include <iostream>
using namespace std;

void odd (int x);
void even (int x);

int main()
{
  int i;
  do {
    cout << "Please, enter number (0 to exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int x)
{
  if ((x%2)!=0) cout << "It is odd.\n";
  else even (x);
}

void even (int x)
{
  if ((x%2)==0) cout << "It is even.\n";
  else odd (x);
}
```

Függvények

- Rekurzió: a függvények esetében a rekurzió azt jelenti, hogy egy függvény a függvénytörzsben önmagát hívja
- Bizonyos feladatoknál ez a fajta megoldás hasznos lehet: javítja az olvashatóságot, könnyebben megkonstruálható egy bizonyos megoldás, stb.
- A rekurzió azonban teljesítményben rosszabb tud lenni a sok függvényhívás miatt

```
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main ()
{
  long number = 9;
  cout << number << "! = " << factorial (number);
  return 0;
}
```

Függvények túlterhelése (function overloading)

- Lehet több függvény is ugyanazzal a névvel, de különböző visszatérési értékkel, paraméter listával (a paraméterek számának vagy legalább egy típusának különböznie kell)
- Ennyi információ alapján a fordító el tudja dönteni, hogy futás időben az átadott argumentumok száma és típusa alapján melyik változatot kell hívni

```
// overloading functions
#include <iostream>
using namespace std;

int operate (int a, int b)
{
  return (a*b);
}

double operate (double a, double b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  double n=5.0,m=2.0;
  cout << operate (x,y) << '\n';
  cout << operate (n,m) << '\n';
  return 0;
}
```

Függvény sablonok (function templates)

- Túlterhelés esetén a függvények törzse megegyezhet és csak a paraméterek típusa különbözik
- Ilyen esetek hatékonyabb kezelésére vezették be a sablonok használatát (generikus típusokkal)
  - `template <template-parameters> function-declaration`
- Figyelni kell, hogy a függvényben használt műveletek definiálva legyenek a megadott típusokban

```
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

Függvény sablonok (function templates)

- A sablon függvény használata során a generikus paraméterek helyére konkrét típusokat vagy értékeket adunk meg:
  - `name <template-arguments> (function-arguments)`
  - A fordító automatikusan példányosítja a megfelelő verziót a függvényből a megadott típusokkal
  - A fordító a legtöbb esetben automatikusan rájön a konkrét generikus típusra az argumentumokból, így azt nem kötelező megadni

```
x = sum<int>(10,20);

x = sum(10,20);
```

Függvény sablonok (function templates)

```cpp
// function template
#include <iostream>
using namespace std;

template <class T>
T sum (T a, T b)
{
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
  cout << k << '\n';
  cout << h << '\n';
  return 0;
}
```

```cpp
// function templates
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
  return (a==b);
}

int main ()
{
  if (are_equal(10,10.0))
    cout << "x and y are equal\n";
  else
    cout << "x and y are not equal\n";
  return 0;
}
```

Függvény sablonok (function templates)

- A sablon paraméterek tartalmazhatnak kifejezéseket is konkrét típusokkal
- Ezek a paraméterek fordítási időben értékelődnek ki, nem futás időben adódnak át a függvénynek
  - Konstans kifejezéseknek kell lenniük (nem lehet pl. változóként átadni)

```
// template arguments
#include <iostream>
using namespace std;

template <class T, int N>
T fixed_multiply (T val)
{
  return val * N;
}

int main() {
  std::cout << fixed_multiply<int,2>(10) << '\n';
  std::cout << fixed_multiply<int,3>(10) << '\n';
}
```

Láthatóság – hatókörök (name visibility – scopes)

- A nevesített entitásokat (változók, függvények, összetett típusok, stb.) deklarálni kell használat előtt. A deklaráció helye hatással van az entitás láthatóságára:
  - Globális láthatóság (global scope): minden blokkon kívül deklarált változó, bárhol a kódban elérhető,
  - Blokk láthatóság (block scope): az entitás egy blokkban (pl. függvény, if-else blokk, stb.) kerül definiálásra és csak azon belül látható (lokális változó)
- Minden hatókörben egy név csak egy változót reprezentálhat

```
int foo;         // global variable

int some_function ()
{
  int bar;       // local variable
  bar = 0;
}

int other_function ()
{
  foo = 1;  // ok: foo is a global variable
  bar = 2;  // wrong: bar is not visible from this function
}
```

```
int some_function ()
{
  int x;
  x = 0;
  double x;   // wrong: name already used in this scope
  x = 0.0;
}
```

Láthatóság – hatókörök (name visibility – scopes)

- Blokk hatókörű változók láthatósága a blokk végéig tart, a blokkon belül lehetnek belső blokkok (inner blocks) is
- A belső blokkon belül ugyanaz a név újra felhasználható (az egy másik blokk), ilyenkor a belső blokk elrejti a külső blokk változóját és a belső blokk változtatásai nem érvényesülnek a külső blokkban

```
// inner block scopes
#include <iostream>
using namespace std;

int main () {
  int x = 10;
  int y = 20;
  {
    int x;   // ok, inner scope.
    x = 50;  // sets value to inner x
    y = 50;  // sets value to (outer) y
    cout << "inner block:\n";
    cout << "x: " << x << '\n';
    cout << "y: " << y << '\n';
  }
  cout << "outer block:\n";
  cout << "x: " << x << '\n';
  cout << "y: " << y << '\n';
  return 0;
}
```

Névterek (namespaces)

- Névterek lehetővé teszik, hogy a nevesített entitások a globális hatókör helyett egy szűkebb ún. névtér hatókörbe kerüljenek
- A névterek használatával a programok különböző elemeit különböző logikai hatókörökbe szervezhetjük és ezeknek a hatóköröknek neveket is adhatunk
  - `namespace identifier { named_entities }`
  - A változók a névtéren belül normál módon érhetők el, a névtéren kívül viszont valamilyen módon hivatkozni kell a névtérre is, amiből használni szeretnénk

```
namespace myNamespace
{
  int a, b;
}
```

```
myNamespace::a;
myNamespace::b;
```

Névterek (namespaces)

- A névterek használatával elsősorban a megegyező nevű entitások közötti ütközéseket tudjuk elkerülni
- A névterek szétbonthatók több egységre, és ezek lehetnek különböző fordítási egységekben (translation units) is (pl. forrás fájlok)

```
namespace foo { int a; }
namespace bar { int b; }
namespace foo { int c; }
```

```
// namespaces
#include <iostream>
using namespace std;

namespace foo
{
  int value() { return 5; }
}

namespace bar
{
  const double pi = 3.1416;
  double value() { return 2*pi; }
}

int main () {
  cout << foo::value() << '\n';
  cout << bar::value() << '\n';
  cout << bar::pi << '\n';
  return 0;
}
```

Névterek (namespaces) - using

- A using kulcsszó segítségével az aktuális hatókörbe „importálhatjuk” a megadott névtér entitásait, így nem kell a teljes nevükkel hivatkozni rájuk
- Több külső csomag használata esetén javasolt a teljes nevek kiírása a jobb olvashatóság miatt

```cpp
// using
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```

```cpp
// using
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

```cpp
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << '\n';
  }
  {
    using namespace second;
    cout << x << '\n';
  }
  return 0;
}
```

Tároló osztályok (storage classes)

- Statikus tároló (static storage):
  - globális vagy névtér hatókörben deklarált változók esetén a tárhelyük a program futásának teljes idejére kerül foglalásra,
  - Azok a változók, amik nem inicalizáltak explicit automatikusan nullák lesznek
- Automatikus tároló (automatic storage):
  - lokális változók esetén a tároló csak abban a blokkban létezik, amelyikhez a változó tartozik és látható, a blokkból való kilépés után a tárhely felhasználható más változók tárolására,
  - Azok a változók, amik nem inicializáltak explicit nem kerülnek inicializálásra automatikusan, értékük meghatározatlan

```
// static vs automatic storage
#include <iostream>
using namespace std;

int x;

int main ()
{
  int y;
  cout << x << '\n';
  cout << y << '\n';
  return 0;
}
```

Feladatok

- Írjunk egy számológép programot, ami a négy alapműveletet képes végrehajtani egyszerre max. két operátorral
  - Tudjon különböző típusú adatokkal dolgozni a program (egész, lebegő pontos)
  - Minden műveletnek legyen egy sablon paraméteres metódusa
  - Az operátorok típusát lehessen bekérni (int, double, stb.)
  - Az operátorokat és a műveletet lehessen bekérni külön-külön vagy egy sorban is (pl.: 3+6)
  - A funkciók legyenek egy saját névtérben
  - A program legyen menü vezérelt, amíg nem lép ki a felhasználó, addig ciklikusan lehessen végrehajtani a műveleteket
  - A felhasználói bemenetek helyességével most nem kell foglalkozni, feltehetjük, hogy mindig megfelelő értékek kerülnek a programba
  - Egészítsük ki a funkcionalitást egy egyszerű memória beépítésével (eredmény hozzáadása a memóriához, eredmény kivonása a memóriából, memória törlése)