# C++

2. HÉT – C++1X FEATURE-ÖK

*A C++ mögött máig aktív szabványosító testület áll – minden 3-4 évben új szabvány jelenik meg.*

*Szerencsére a 'gyártók' (akik a fordítókat készítik) a szabványt komolyan is veszik.*

AZ ELSŐ NÉHÁNY HÉTBEN A C++11 ÉS AZÓTA MEGJELENT ÚJÍTÁSOKRÓL ESIK SZÓ

# Swap1 és Swap2 profilozása

A legutóbbi alkalommal szó esett a jobb oldali referenciákról, és arról, hogy a jobb oldali referenciák használata lényegében hatékonysági kérdés, mivel a jobb oldali referencia használata a felesleges másolások helyett “mozgatást” eredményez

- Természetesen a szimpla bal oldali referenciát is használhatjuk hatékonysági okokból, csak ilyenkor egy korábbi változó alól biztosan nem “húzzuk ki” a dinamikus memóriát – túl azon, hogy a bal oldali referenciát arra is használhatjuk, hogy egy másik scope-on belül, pl. függvényen belül módosíthassuk egy “külső” változó értékét.
- Mozgatás alatt viszont azt értjük, hogy egy adott címen levő változó értékét ezentúl nem fogjuk használni, ezért annak dinamikus részei “ellophatóak” egy másik változó részére

Mit jelent ez hatékonyságban? Profilozzuk a két swap-változatot, amit múlt orán néztünk!

# swap1 és swap2 profilozása

```
struct FunctionTimer {

    std::map<std::string, float> functionTimes;

    int numTimesToRun;


    FunctionTimer(int timesToRun) : numTimesToRun{timesToRun} {}


    template <typename Function>

    void profileFunction(Function f, std::string nameOfFunction) {

        clock_t startTime = clock();

        for (int iteration = 0; iteration < numTimesToRun;
iteration++) { f(); }

        clock_t endTime = clock();

        float runtime = (endTime - startTime) / static_cast<float>(CLOCKS_PER_SEC);

        functionTimes[nameOfFunction] = runtime;

    }


    void report() {

        for (auto pairOfItems : functionTimes) {

            std::cout << pairOfItems.first << ": " << pairOfItems.second << std::endl;

        }

    }

};
```

# swap1 és swap2 profilozása

```
void swap1(std::string& a, std::string& b) {
    std::string temp{ a };
    a = b;
    b = temp;
}


class FunctorSwap1 {
    std::string a, b;
public:
    FunctorSwap1() : a("string1"), b("string2") {}
    void operator()() { swap1(a, b); }
};
```

```
void swap2(std::string& a, std::string& b) {
    std::string temp{ std::move(b) };
    b = std::move(a);
    a = std::move(temp);
}
class FunctorSwap2 {
    std::string a, b;
public:
    FunctorSwap2() : a("string1"), b("string2") {}
    void operator()() { swap2(a, b); }
};
```

# swap1 és swap2 profilozása

Innentől a profilozás már gyerekjáték:

```
int main()
{
    FunctionTimer ft(100 * 1000 * 1000 * 1000 * 1000); // 100 trillion
    ft.profileFunction(FunctorSwap1(), "swap1");
    ft.profileFunction(FunctorSwap2(), "swap2");
    ft.report();
    std::cin.get();
    return 0;
}
```

# swap1 és swap2 profilozása

A kód egyben megtekinthető ezen a coliru linken: http://coliru.stacked-crooked.com/a/6dd64d854171fd88

Az –O2 kapcsoló gyorsabbá teszi a futást, az –O3 pedig swap használat esetén még többet optimalizál:

```
g++ -std=c++17 -O2 -Wall -pedantic -pthread main.cpp && ./a.out
Timing the two functions
swap1: 0.040715
swap2: 0.023068
```

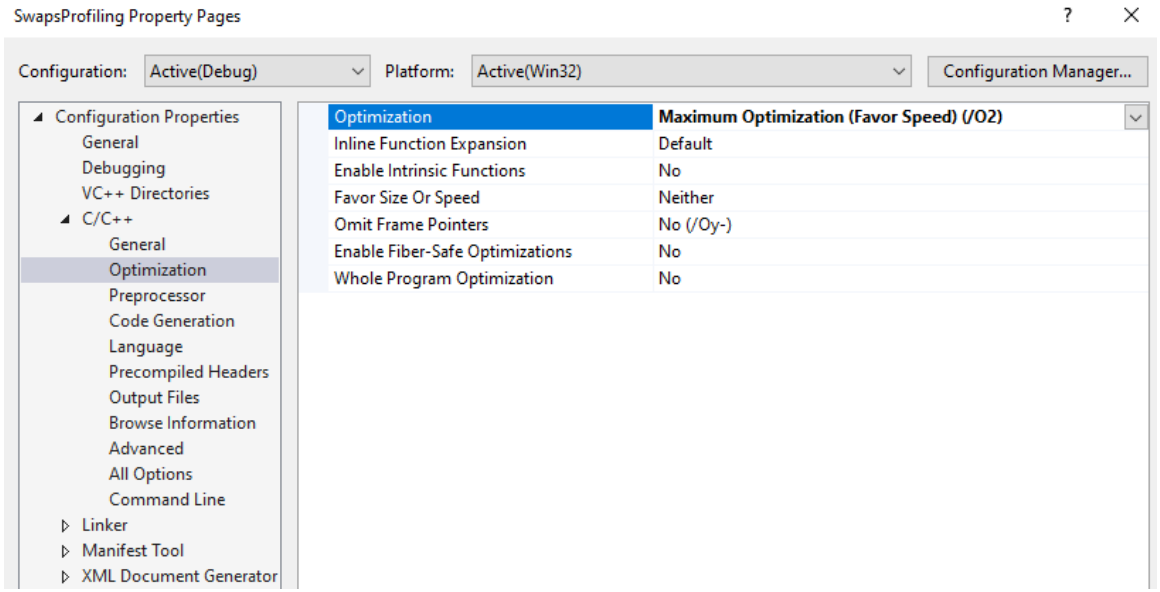
```
Timing the two functions
swap1: 0.044918
swap2: 0.014946
```

```
g++ -std=c++17 -O3 -Wall -pedantic -pthread main.cpp && ./a.out
```

# swap1 és swap2 profilozása

Visual Studio 2017-ben külön ellenőrizni kell az optimalizációt, nehogy ‘disabled’ legyen

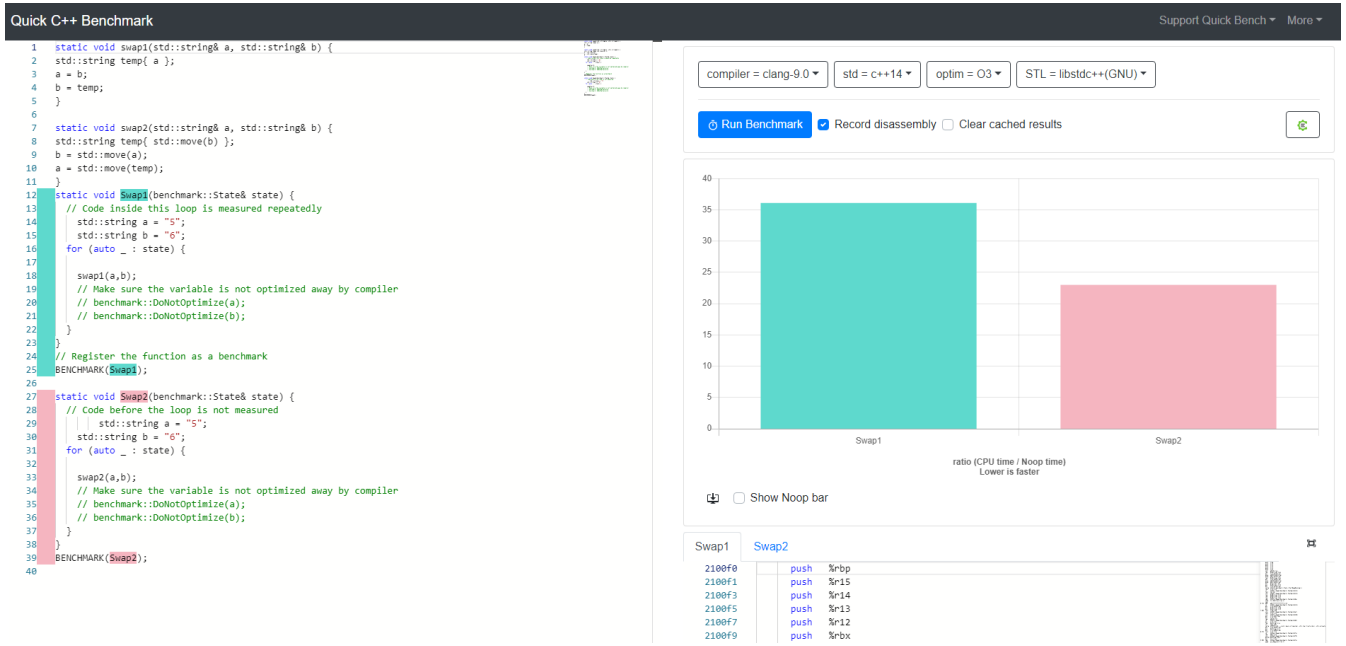
Emellett ügyeljünk arra, hogy nem debug módban futtatjuk a programot (ami járulékos költségekkel jár)



# swap1 és swap2 profilozása

Érdemes ezt az online tool-t is megismerni:

http://quick-bench.com/unkft0RzrUDFL1tdnKqh7rme7y4



# Lambda függvények

Sokszor bosszantó, hogy miért kell egy külön functor osztályt készíteni csak azért, hogy egyetlen függvényhívásnak átadjuk.

Naggyá és átláthatatlanná is teszi a kódot, ha sok feleslegesfunctort kell írnunk.

A C++11-ben ilyen esetekre vezették be a **lambda függvényeket.** Ezek felfoghatóak úgy, mint temporary, név nélküli függvények.

A lambda függvények szintaxisa a következő:

```
[ captures ] (parameters) -> returnTypesDeclaration { lambdaStatements; }
```

A capture-ök lehetővé teszik, hogy a lambda függvényen belül felhasználhassuk (akár másolatként, akár referenciaként a környezet bizonyos változóit. Természetesen a parameter-listában is lehet referencia, csak ezt mindig át kell adni, amikor meghívjuk a függvényt. A “capture” ezzel szemben egyfajta hiperparamétere a függvénynek, a környezet alapján.

# Lambda függvények

Egy példa, ahol a capture és paraméter lista közötti különbség számít:

```
std::vector<int> items;
int factor;
auto foundItem = std::find_if(items.begin(), items.end(),
[&factor](int const& a)
{
   return a * factor == 100;
});
```

Mivel foundItem()-et az std::find_if fogja meghívni, nem tudjuk biztosítani, hogy minden meghívásakor a factor értékét is átadja, mint argumentumot!

# Lambda függvények

A legegyszerűbb esetben nincsen se capture, se argumentum, se return type (az utóbbi kettő helyére opcionálisan be lehet írni egy “void”-ot)

Ha nincsen return type, a nyíl elhagyható, ahogy már az előző fólián is láthattuk.

Az olyan lambda függvények, amiknek nincs capture-ük, funkcionális értelmeben “pure” (“tiszta”) függvények – eredményük csak az átadott argumentumoktól (paraméterektől) függ!

```
auto lambda = []() { std::cout << "..." << std::endl; };

// ekvivalens:

auto lambda2 = [](void) -> void { std::cout << "..." << std::endl; };

lambda();
```

# Lambda függvények

A capture-öket rengetegféleképpen meg lehet adni:

`[a, &b]`—Lambda head from the previous example; `a` is captured by value, and `b` is by reference.

`[]`—A lambda that doesn't use any variable from the surrounding scope. These lambdas don't have any internal state and can be implicitly cast to ordinary function pointers.

`[&]`—Captures all variables that are used in the lambda body by reference.

`[=]`—Captures all variables that are used in the lambda body by value.

`[this]`—Captures the `this` pointer by value.

`[&, a]`—Captures all variables by reference, except `a`, which is captured by value.

`[=, &b]`—Captures all variables by value, except `b`, which is captured by reference.

# Lambda függvények

Például, az előző példában elhagyhatóak lettek volna a functorok:

```
int main()
{
    FunctionTimer ft(100 * 1000 * 1000 * 1000 * 1000);
    // 100 trillion
    int a = 5;
    int b = 6;
    // ft.profileFunction(FunctorSwap1(), "swap1");
    // ft.profileFunction(FunctorSwap2(), "swap2");
    ft.profileFunction([&a, &b]()->void {swap1(a, b); }, "swap1");
    ft.profileFunction([&a, &b]()->void {swap2(a, b); }, "swap2");
    ft.report();
    std::cin.get();
     return 0;
}
```

# Lambda függvények

A lambda függvények hasznosak, de fontos tudni, hogy nem egy új funkciót jelentek, csak “szintaktikai édesítőszerként” működnek. Semmi olyan dolog nincs C++-ban, amit lambda függvényekkel meg lehet oldani, de nélkülük nem.

A lambda függvények bár tömörek, vannak olyan esetek, amikor még lambda függvényt írni is feleslegesen hosszú. Az STL könyvtárban számos olyan függvény található (std::accumulate, std::sort), ahol a viselkedést a megfelelő operator megválasztásával módosítani lehet. Ilyen esetekben hasznosak az alábbi beépített függvények:

# Lambda függvények

| Group | Wrapper name | Operation |
|---|---|---|
| Arithmetic operators | `std::plus` | `arg_1 + arg_2` |
| | `std::minus` | `arg_1 - arg_2` |
| | `std::multiplies` | `arg_1 * arg_2` |
| Arithmetic operators (continued) | `std::divides` | `arg_1 / arg_2` |
| | `std::modulus` | `arg_1 % arg_2` |
| | `std::negates` | `- arg_1` (a unary function) |

# Lambda függvények

| | | |
|---|---|---|
| Comparison operators | `std::equal_to` | `arg_1 == arg_2` |
| | `std::not_equal_to` | `arg_1 != arg_2` |
| | `std::greater` | `arg_1 > arg_2` |
| | `std::less` | `arg_1 < arg_2` |
| | `std::greater_equal` | `arg_1 >= arg_2` |
| | `std::less_equal` | `arg_1 <= arg_2` |
| Logical operators | `std::logical_and` | `arg_1 && arg_2` |
| | `std::logical_or` | `arg_1 \|\| arg_2` |
| | `std::logical_not` | `!arg_1` (a unary function) |
| Bitwise operators | `std::bit_and` | `arg_1 & arg_2` |
| | `std::bit_or` | `arg_1 \| arg_2` |
| | `std::bit_xor` | `arg_1 ^ arg_2` |

# Meghívható egyedek típusai

Láthattuk, hogy számos módja van a meghívható egyedek (callables) definiálásának. Néhány opció:

- függvény pointerek
- függvény objektumok (functorok)
- függvény pointerre konvertálódó objektumok (függvény pointer operator, vagyis “operator function_ptr(){}”, ami egy function_ptr típust ad vissza, és ezért alkalmas az objektum function_ptr típusúra konvertálására)
- lambda függvények
- beépített, nevesített függvények (std::multiplies, stb.)

A fő kérdés most, hogy milyen típussal jellemezhetjük ezeket generikusan? Tegyük fel, hogy egy osztály tagjaként el akarunk menteni egy vagy több meghívható egyedet, de mindegy nekünk, hogy ezek pontosan milyen típusú meghívható egyedek.

# Meghívható egyedek típusa

Erre nyújt megoldást a C++11 óta az std::function típus. Ez egy template-típus, ahol a template-ben először a visszatérési típust kell megadni, majd zárójelben a paraméter(ek) típusát:

```
std::function<void(int,int)>>
```

Ez itt egy olyan meghívható egyed (callable), ami két int-et vár és semmit nem ad vissza. De ezen felül mindegy, hogy függvény pointer vagy functor vagy micsoda.

# Meghívható egyedek típusa

A korábbi példában megtehettük volna, hogy eltároljuk az összes függvényt, majd egyszerre profilozzuk őket:

```
struct FunctionTimer {
    std::vector<std::function<void(void)>> functions;
    std::vector<std::string> names;
    std::vector<float> times;
    int numTimesToRun;
    FunctionTimer(int timesToRun) : numTimesToRun{ timesToRun } {}
    void addFunction(std::function<void(void)> f, std::string name) {
        functions.push_back(f); names.push_back(name); times.push_back(0.0);
    }
    void profileFunctions() {
        for (int inx = 0; inx < functions.size(); inx++) { clock_t startTime = clock();
            for (int iteration = 0; iteration < numTimesToRun; iteration++) { functions.at(inx)(); }
            clock_t endTime = clock();
            float runtime = (endTime - startTime) / static_cast<float>(CLOCKS_PER_SEC);
            times[inx] = runtime;
        }
    }
    void report() {
        for (int i = 0; i < times.size(); i++) { std::cout << names[i] << ": " << times[i] << std::endl; }
    }
};
```

# Meghívható egyedek típusa

Vegyük észre, hogy ha std::function helyett függvény pointert használunk, akkor functorral és lambdával nem lenne használható a FunctionTimer struct:

```
typedef void(*Fp)();


struct FunctionTimer {
    std::vector<std::function<void(void)>> functions;
    std::vector<Fp> functions2;
    std::vector<std::string> names;
    std::vector<float> times;

    // ...
```