constexpr და noexcept

C++ -ში ისეთი ობიექტები, რომლების მნიშვნელობა ცნობილია კომპილაციის დროს, შესაძლოა მოთავსდნენ მეხსიერების ისეთ ნაწილში, საიდანაც მათი წაკითხვა უფრო სწრაფად მოხერხდება. ამიტომ მნიშვნელოვანია მათი მონიშვნა სპეციალური სიტყვებით (`constexpr`, `const`). ამას გარდა, ცნობილია რომ ჩვეულებრივი C-ის ტიპის მასივები და array<> კონტეინერები საგრძნობლად სწრაფია ვიდრე ვექტორები, სიები და სხვა კონტეინერები. თუმცა მათი შექმნისას საჭიროა ელემენტების რაოდენობის მითითება. სტანდარტის მიხედვით, ელემენტების რაოდენობა უნდა იყოს მუდმივი. ეს ქმნის გარკვეულ პრობლემებს (იხ. ქვემოთ), რისი გადაჭრაც უკვე შესაძლებელია `constexpr` -ის დახმარებით.

`constexpr` ობიექტები არიან აგრეთვე `const`, და მათ აქვთ მნიშვნელობები, რომლებიც ცნობილია კომპილაციის დროს. თუმცა შებრუნებული დებულება სამართლიანი არაა. განვიხილოთ მაგალითები:

```
 int sz;                          // არა-constexpr ცვლადი
constexpr auto n = sz;            //  შეცდომაა!  sz-ის  მნიშვნელობა  არაა
                                  // ცნობილი
                                  // კომპილაციის დროს
std::array<int, sz> a;            // შეცდომაა, იგივე მიზეზის გამო
constexpr auto m = 10;            // სწორია
std::array<int, m> b;             // სწორია
```

`const` -ის მნიშვნელობებს არ მოეთხოვებათ ამდენი. მაგალითად:

```
int sz;
const auto n = sz;                // სწორია
std::array<int, n> a;             // შეცდომაა! sz-ის მნიშვნელობა არაა
                                  // ცნობილი
```

// კომპილაციის დროს როგორც ვხედავთ, ზოგიერთი `const` არ არის `constexpr`.

`constexpr` ფუნქციების თაობაზე ცნობილია, რომ:

- ვთქვათ, `constexpr` ფუნქცია გამოიყენება როდესაც საჭიროა კომპილაციის-დროინდელი მუდმივის დაბრუნება. თუ ასეთი ფუნქციებისთვის გადაწოდებული არგუმენტების მნიშვნელობები ცნობილია კომპილაციის დროს, შედეგიც გამოითვლება კომპილაციის დროს. თუ რომელიმე არგუმენტის მნიშვნელობა არაა ცნობილი კომპლილაციის დროს, კოდი უარყოფილ იქნება. მაგალითად, არ იმუშავებს

```
auto n = 5;


array<int,factorial(n)>a;
```

- მეორე მხრივ, ეს ფუნქცია მუშაობს როგორც ჩვეულებრივი ფუნქცია და არ არის მისი გადატვირთვა საჭირო `constexpr` -სიტყვის გარეშე.

C++11-ში, `constexpr` ფუნქცია ძალიან შეზღუდულია. მოითხოვება რომ მისი ტანი არ უნდა იყოს ერთ შესრულებად შეტყობინებაზე მეტი. ამ შეზღუდვას რეკურსიის გამოყენებით ვუვლით გვერდს. C++14 თითქმის არ ზღუდავს ასეთ ფუნქციებს, მაგრამ ყველა კომპილერი არაა ამ დონეზე აყვანილი, ჯერ-ჯერობით. `constexpr` ფუნქციები შეზღუდულია იმითაც, რომ არგუმენტად უნდა მიიღონ და უნდა დააბრუნონ ე.წ. ლიტერალური ტიპები (*literal type*).

`Constexpr` შეგვიძლია გამოვიყენოთ რიგი ფუნქციების აღსაწერად. მაგალითად:

```
//გრადუსების რადიანებში გადაყვანა
constexpr
double degreeToRadian(double deg) noexcept
{
return (deg*Pi) / 180.0;
}
```

ან კიდევ და სიმბოლოების მთვლელი:

```
//დაბალი რეგისტრის სიმბოლოების მთვლელი
constexpr int countlower(const string & s, int n = 0, int c = 0) noexcept
{
	return n == s.size() ? c :
		'a' <= s[n] && s[n] <= 'z' ? countlower(s, n + 1, c + 1) :
		countlower(s, n + 1, c);
}
```

სხვა მაგალითები შეგიძლიაქ იხილოთ აქ.

C++98-ში განსაკუთრებული შემთხვევების ანუ გამონაკლისების დახასიათებები ცვალებადობით გამოირჩეოდნენ. თქვენ გიწევდათ დაგედგინათ იმ გამონაკლისების ტიპი, რომელიც შესაძლოა გამოეყო ფუნქციას. ამგვარად, თუ იმპლემენტაცია განიცდიდა ცვლილებას, გამონაკლისის დახასიათებასაც შესაძლოა დასჭირვებოდა ცვლილება. გამონაკლისის დახასიათების ცვლილებამ შესაძლოა დააზიანოს მომხმარებლის კოდი, რადგან გამომძახებლები შესაძლოა დამოკიდებული ყოფილიყვნენ გამონაკლისის საწყის დახასიათებაზე. კომპილერები ჩვეულებრივ არ გვთავაზობდნენ რაიმე დახმარებას ფუნქციების იმპლემენტაციების, გამონაკლისების დახასიათების და მომხმარებლის კოდის თავსებადობის შენარჩუნებაში. პროგრამისტების უმეტესობა საბოლოოდ მიდიოდა დასკვნამდე რომ C++98-ის გამონაკლისების დახასიათება არ ღირდა თავის შეწუხებად.

C++11-ის სტანდარტზე მუშაობისას მიაღწიეს შეთანხმებას, რომ ფუნქციის გამონაკლისების გამოყოფის შესახებ ნამდვილად მნიშვნელოვანი ინფორმაცია იმაში მდგომარეობს, აქვს თუ არა მას რაიმე გამონაკლისი (განსაკუთრებული შემთხვევა). შავი ან თეთრი, ფუნქციამ შეიძლება გამოყოს განსაკუთრებული შემთხვევები ან იგი ამას გარანტირებულად არ იზამს. „შეიძლება ან არასდროს“ დიხოტომიამ შექმნა C++11-ის გამონაკლისების დახასიათების საფუძველი, რომელმაც არსებითად შეცვალა C++98-ისა. (C++98-ის გამონაკლისების დახასიათებები ისევ ძალაშია, თუმცა მათი გამოყენება არაა წახალისებული/რეკომენდირებული.) C++11-ში უპირობო `noexcept` განკუთვნილია იმ ფუნქციებისთვის, რომლებიც გარანტირებულად არ გამოყოფენ განსაკუთრებულ შემთხვევებს.

ინტერფეისის შემუშავების საკითხია უნდა იყოს თუ არა ფუნქცია ასეთად გამოცხადებული. ფუნქციის გამონაკლისების-გამომყოფი ყოფაქცევა საკვანძო მნიშვნელობისაა მომხმარებლებისთვის. გამომძახებლებს შეუძლიათ მოითხოვონ ფუნქციის noexcpet სტატუსი და ასეთი მოთხოვნის პასუხს შეუძლია იმოქმედოს გამონაკლისის დაცულობაზე ან გამომძახებელი კოდის ეფექტურობაზე. როგორც ასეთი, ფუნქციის noexcept-ობა ისეთივე მნიშვნელოვანი მონაცემია როგორც წევრი ფუნქციის const-ობა. თუ თქვენ იცით რომ ფუნქცია არ გამოყოფს გამონაკლისებს მაგრამ განაცხადში გამოტოვებულია noexcept, ე.ი. გვაქვს ინტერფეისის ცუდი დახასიათება.

არსებობს სხვა სტიმული თუ რატომ უნდა გამოვაცხადოთ ფუნქცია noexcept-ად როდესაც ის არ გამოყოფს გამონაკლისებს: ის ნებას რთავს კომპილერს, რომ შექმნას უკეთესი ობიექტური კოდი. იმის გაგებაში თუ რატომ, დაგვეხმარება განსხვავებების შესწავლა იმ გზებს შორის, რომლითაც C++98 და C++11 ამბობენ რომ ფუნქცია არ გამოყოფს გამონაკლისს. განვიხილოთ *f* ფუნქცია, რომელიც პირდება გამომძახებელს, რომ მისგან ვერასდროს მიიღებს გამონაკლის შემთხვევას. ამის გამოსახატად არსებობს ორი გზა:

```
int f(int x) throw();   // არავითარი გამონაკლისი f-ისგან: C++98
int f(int x) noexcept;  // არავითარი გამონაკლისი f-ისგან: C++11
```

თუ მუშაობის პროცესში გამონაკლისი ტოვებს *f* -ს, *f*-ის გამონაკლისის დახასიათება ირღვევა. C++98-ის მიერ გამონაკლისის დახასიათების შემთხვევაში, გამომძახებების სტეკი ჩახსნილია *f*-ის გამომძახებელისგან და, ზოგიერთი შეუფერებელი მოქმედების შემდეგ, პროგრამის შესრულება წყდება. C++11-ის მიერ გამონაკლისის დახასიათების შემთხვევაში, პროგრამის მსვლელობა ოდნავ განსხვავებულია: პროგრამის შესრულების შეწყვეტის წინ სტეკი მხოლოდ შეძლებიდაგვარად არის ჩახსნილი.

განსხვავებას გამომძახებების სტეკის ჩახსნასა და შეძლებისდაგვარად ჩახსნას შორის აქვს გასაოცრად დიდი გვლენა კოდის წარმოქმნაზე. noexcept ფუნქციაში, ოპტიმაიზერებს არ სჭირდებათ შესრულების სტეკის შენარჩუნება ჩახსნილ მდგომაროებაში, თუ გამონაკლისი გამოდის ფუნქციიდან, არც იმას უზრუნველყოფენ რომ ობიექტები noexcept ფუნქციაში დაიშლებიან შექმნის შებრუნებული მიმდევრობით თუ გამონაკლისი ტოვებს ფუნქციას. ფუნქციებს გამონაკლისის “throw()” დახასიათებით აკლიათ ასეთი ოპტიმიზაციის მოქნილობა, ისევე როგორც ფუნქციებს საერთოდ გამონაკლისის დახასიათების გარეშე. ვითარება შესაძლოა შევაჯამოთ ამგვარად:

```
RetType function(params) noexcept;          // მეტად ოპტიმიზებული

RetType function(params) throw();           // ნაკლებად ოპტიმიზებული

RetType function(params);                   // ნაკლებად ოპტიმიზებული
```

მხოლოდ ეს მიზეზიც საკმარისია იმისთვის რომ გამოვაცხადოთ ფუნქცია noexcept-ად როდესაც იცით რომ ის არ წარმოშობს გამონაკლისებს.

ახლა დავაკვირდეთ ზემოხსენებული სიტყვაგასაღებების გამოყენებით როგორი განსხვავებებია პროგრამის შესრულების დროში. შევქმნათ რამე მარტივი ხელოვნური ფუნქცია თვალსაჩინოებისთვის:

```
int fo(int a)
{
	if (a == 1) return 1;
	return a+ fo(a - 1);}
```

პროგრამა დრაივერს მივცეთ შემდეგი სახე:

```
int main()
{
	constexpr int  m = 111;
	int t = 0;

	steady_clock::time_point begin = steady_clock::now();
	for (int i = 0; i < 1000000; ++i)
	{
		t = fo(m);
	}
	steady_clock::time_point end = steady_clock::now();
	cout << "microsecconds  = " <<
		duration_cast<microseconds>(end -begin).count()<< endl;
	cin >> t;
	return 0;

}
```

პროგრამა დაგვიბეჭდავს fo(T) გამოძახებაზე დახარჯულ დროს მიკროწამებში(ციკლის ოპერაციებზე დახარჯული დრო შეგვიძლია უგულებელვყოთ);

ხელწერა	შედეგი
`int fo(int a)`	`// microseconds = 2957112`
`constexpr int fo(int a)`	`// microseconds = 2883095`
`int fo(int a) noexcept`	`// microseconds = 2852256`
`constexpr int fo(int a) noexcept`	`// microseconds = 2839266`

თუ `constexpr` ფუნქციებს გამოვაცხადებთ `noexcept`-ად ეს იქნება ყველაზე ოპტიმიზირებული კოდი, მაგრამ უნდა გავითვალისწინოთ ერთი გარემოება რომ `noexcept`-ად გამოცხადებულ ფუნქციაში თუ ვიძახებთ სხვა ფუნქციას ისიც უნდა იყოს `noexcept`. შეიძლება არ იყოს გამოცხადებულ `noexcept`-ად მაგრამ მაშინ გარანტირებულად უნდა ვიცოდეთ რომ არ გამოიტყორცნის გამონაკლისი შემთხვევის შეტყობინებას, რომ მივიღოთ სასურველი ეფექტურობა. წინააღმდეგ შემთხვევაში მივიღებთ განუზღვრელ ყოფაქცევას.

`noexcept` და `constexpr` სიტყვაგასაღებები შეგვიძლია გამოვიყენოთ ობიექტზე ორიენტირებულ პროგრამირებაში. ანუ შეგვიძლია გვქონდეს `constexpr` და `noexcept` კონსტრუქტორი და მეთოდები. მაგალითად ვნახოთ კომპლექსური რიცხვის მარტივი წარმოდგენა:

```
//კომპლექცური რიცხვის წარმოდგენა
struct complex
{
	//a constant expression constructor
	constexpr complex(double r, double i) noexcept : re(r), im(i) { }
//empty body
	//constant expression functions
	constexpr double real() noexcept{ return re; }
	constexpr double imag() noexcept { return im; }
private:
	double re;
	double im;
};
```

ახლა განვიხილოთ ისეთი მონაცემთა სტრუქტურა როგორიცაა Segment Tree. ეს მონაცემთა სტრუქტურა გამოიყენება როცა გვაინტერესებს დიაპაზონის რაღაც თვისება, მაგალითად მინიმალური ან მაქსიმალური ელემენტი (i,j) დიაპაზონში, დიაპაზონის ჯამი და ასე შემდეგ. ჩვენ განვიხილოთ Segment Tree-ს ისეთი იმპლემენტაცია როცა ის გვიბრუნებს დიაპაზონში ელემენტების ნამრავლის ნიშანს 0, + ან -. Segment Tree-ში გამოყენებული ფუნქციები თითქმის ყველა რეკურსიულია, ამიტომ აშკარაა რომ მათი noexcept-ად გამოცხადება უფრო ეფექტურს გახდის ჩვენს კოდს. ვნახოთ შესაბამისი იმპლემენტაცია:

```
#pragma once
#include<vector>
#include<algorithm>

using namespace std;

class my_segment_tree
{
public:
	my_segment_tree(const vector<int>& input)noexcept;
	int query(int qlow, int qhigh)noexcept;
	void update_node(int index, int value)noexcept;



private:
	int next_pow_2(int n)noexcept;
	int left(int node)noexcept;
	int right(int node)noexcept;
	void build(const vector<int>& input, int node, int low,
	                                                  int high)noexcept;
	int range_minimum_query(int qlow, int qhigh, int low,
	                                        int high, int node)noexcept;
	void update_node(int node, int low, int high, int index,
	                                                    int value)noexcept;
	vector<int> st;
	int H;

};
```

```
//კონსტრუქტორი
my_segment_tree::my_segment_tree(const vector<int>& input) noexcept
{
	int size = next_pow_2(input.size()) * 2;
	H = input.size() - 1;  // შემოსული ვექტორის სიგრძე
	for (int i = 0; i < size; i++)
	{
		st.push_back(1);	//გამრავლების მიმართ ერთეულოვანი ელემენტია 1
	}
	build(input, 1, 0, H);

}
//მოთხოვნა დიაპაზონის ნიშნის დასადგენად
int my_segment_tree::query(int qlow, int qhigh) noexcept
{
	return range_minimum_query(qlow, qhigh, 0, H, 1);
}

//კვანძის განახლება ღია ფუნქცია
void my_segment_tree::update_node(int index, int value) noexcept
{
	update_node(1, 0, H, index, value);
}
//კვანძის განახლება პრივატული ფუნქცია
void my_segment_tree::update_node(int node, int low, int high, int index, int
value) noexcept
{
	if (low == high)
		st[node] = value;
	else
	{

		int mid = (low + high) / 2;
		if (index <= mid)
		{
			update_node(left(node), low, mid, index, value);
		}
		else
		{
			update_node(right(node), mid + 1, high, index, value);
		}
		st[node] = st[left(node)] * st[right(node)];
	}
}
//რეკურსიული მოთხოვნა პრივატული
int my_segment_tree::range_minimum_query(int qlow, int qhigh, int low, int
high, int node) noexcept
{
	if (qlow <= low && qhigh >= high)
		return st[node];
	if (qlow > high || qhigh < low)
		return 1;
	int mid = (low + high) / 2;
```

```
	return range_minimum_query(qlow, qhigh, low, mid, left(node)) *
		range_minimum_query(qlow, qhigh, mid + 1, high, right(node));

}
// n-ის შედეგი 2-ის ხარისხი საცავი ვექტორის ზომის დასადგენად
int my_segment_tree::next_pow_2(int n) noexcept
{

	int k = 1;
	while (k < n)k <<= 1;
	return k;
}

//st ვექტორის აშენება
void my_segment_tree::build(const vector<int>& input, int node,
                                                    int low, int high) noexcept
{
	if (low == high)
	{
		st[node] = input[low];

	}
	else
	{
		int mid = (low + high) / 2;
		build(input, left(node), low, mid);  //მარცხენა ქვეხის აშენება
		build(input, right(node), mid + 1, high);  // მარჯვენა ქვეხის აშენება
		st[node] = st[left(node)] * st[right(node)];    // კვანძში
                                                                // მნიშვნელობის ჩასმა
	}

}
//მარცხენა შვილი
int my_segment_tree::left(int node) noexcept
{
	return node << 1;
}
//მარჯვენა შვილი
int my_segment_tree::right(int node) noexcept
{
	return ((node << 1) + 1);
}

//რადგან მხოლოდ ნიშანი გვაინტერესებს ამიტომ მონაცემების
//კონვერტირება უფრო მარტივ რიცხვებზე უკეთესია
//ნამრავლებისთვის რომ გვეყოს int
int convert(int n) noexcept
{
	if (n > 0)return 1;
	else if (n < 0) return -1;
	return 0;
}
```

როგორც ვხედავთ ყველა მეთოდი გამოვაცხადეთ `noexcept` -ად. თუ გავიხსენებთ ზემოხსენებულ მაგალითს თუ რამდენად ამცირებს შესრულების დროს `noexcept` სიტყვაგასაღები აშკარა გახდება რომ ასე იმპლემენტირებული Segment Tree-ის შესრულების დრო ბევრად ნაკლები იქნება ჩვეულ იმპლემენტაციასთან შედარებით. ***ამიტომ `noexcept` ყველგან და ყოველთის როცა ეს შესაძლებელია.***

მაგალითები უფრო დაწვრილებით შეგიძლიათ იხილოთ აქ.

წყარო:

1. https://kobage.info/ArticlePositions/GetArticlePositionFile/81
2. https://kobage.info/ArticlePositions/GetArticlePositionFile/115
3. Effective Modern C++, by Scot Meyers
4. https://msdn.microsoft.com/en-us/library/dn956974.aspx
5. https://msdn.microsoft.com/en-us/library/dn956976.aspx