

C/C++ কোডিং টিপস অপটিমাইজড প্রোগ্রামিং-এর কৌশল

উদীয়মান নতুন প্রোগ্রামারদের উদ্দেশ্যেই এই লেখা। আপনারা এসিএম প্রোগ্রামিং কনটেস্টের প্রবলেমগুলোতে দেখবেন, সেখানে টাইম এবং মেমোরি এফিশিয়েন্ট সল্যুশন চাওয়া হয়। আপনি হয়তো সমস্যাটি সমাধান করছেন কিন্তু কিছুতেই জাজ কর্তৃক গৃহীত হচ্ছে না। তাহলে নিশ্চয়ই আপনার মনে প্রশ্ন জেগেছে কিভাবে আরো দ্রুতগতির ও ভালো টেকনিক ব্যবহার করা যায়? এখানে কিছু অতি সাধারণ ধরনের অপটিমাইজেশনের কৌশল দেখানো হবে, যার অনুশীলন আপনাকে ভালো কোডিং-এ অনুপ্রাণিত করবে।

সাধারণত কোড অপটিমাইজেশন-এর ক্ষেত্রে সময়টাই প্রধানত বিবেচনা করা হয়। অনেকক্ষেত্রেই আমরা ব্যস্ত হই ডিকম্পেস অথবা মেমোরি লিমিট-এর ব্যাপারে। সেটা অনেকাংশে ফলায়ক ও বাটে। কিন্তু DOS এর যুগের সেই ৬৪০ কি. বা এর সমস্যা এখন আর নেই। তবে এখন প্রসেসর টাইম সর্বাপেক্ষা গুরুত্বপূর্ণ ব্যাপার। পিসির বেলায় একথা প্রযোজ্য— তবে এমবেডেড সিস্টেম যেমন যেসব পিডিএ বা পামপাইলট প্রভৃতিতে উইন্ডোজ সিই রয়েছে, তাতে মেমোরি আর কোড সাইজ অবশ্যই গুরুত্বপূর্ণ।

আমরা এখানে কিছু গুরুত্বপূর্ণ অপটিমাইজেশন মেথড-এর আলোচনা করব। এগুলো আপনার কোডকে আরো ফাস্ট রানিং-এ সহায়তা করবে। অপটিমাইজেশন আর কিছুই না— ক্ষুদ্র ক্ষুদ্র পরিবর্তন বা পরিমার্জন যা কিছু কিছু মাইক্রোসেকেন্ড করে সাশ্রয় করবে। আর জানেন তো 'বিন্দু বিন্দু জলই গড়ে তোলে সাগর অতল। এসব অপটিমাইজেশন যখন কোনো লুপের ভেতর থাকে বা অনেকবার একই ফাংশন কল হয়ে থাকে, তখন পরিবর্তনের সুফলটা স্পষ্ট হয়। আর আপনি যদি ভাবেন— আপনার প্রজেক্টটি খুব ছোট আকারের এতে আবার অপটিমাইজেশন-ফপটিমাইজেশন কী দরকার? তাহলে ভুল করবেন। কেননা ভালো কোডিং পকটিস সবসময়ই জরুরি।

আমরা শুরু করব সহজে শেখা যায় বা প্রয়োগ করা যায় এমন ধরনের কোডিং দিয়ে।

দ্রুত ক্যালকুলেশন

কম্পিউটার কাজ করে বাইনারি সংখ্যা নিয়ে, আর আমরা ডেসিমাল যা ক্যালকুলেশনকে মন্থর করে দেয়। আমরা কিছু কিছু মন্থরগতির ক্যালকুলেশনকে বাইনারিতে পরিবর্তন করে একে ত্বরান্বিত করতে পারি।

ইন্টিজারের পূর্ণ বর্গমূল

কোনো ইন্টিজার-এর বর্গমূল বের করার জন্য, ঐ ইন্টিজার থেকে ক্রমাগত বিজোড় সংখ্যাধারাকে বিয়োগ করতে হবে। যতক্ষণ না ফলাফল শূন্য বা তার কম না হয়। এজন্য যতবার বিয়োগ করতে হবে তা-ই ঐ ইন্টিজারটির ইন্টিজার বর্গমূল। এ প্রক্রিয়াটি অবশ্যই সাধারণভাবে বর্গমূল নির্ণয়ের প্রক্রিয়ার চেয়ে দ্রুততর। তবে এটি কেবল ইন্টিজারের ইন্টিজার বর্গমূলের জন্যই প্রযোজ্য। একটি সংখ্যা ৯ নেয়া যাক—

9

9 - 1 = 8

8 - 3 = 5

5 - 5 = 0

3 (number of subtractions) is the integer square root of 9.

Multiplication and Division by a Power of 2

নিশ্চয়ই বুঝতে পারছেন বিট শিফটিং-এর কথা বলতে যাচ্ছি। কেননা বিট শিফটিং করে অনেক দ্রুত গুণ ও ভাগের কাজ করা যায়, তবে ভাজককে দুইয়ের পাওয়ার হতে হবে। আপনারা অনেকের কাছে এটা তেমন নতুন নয়।

$x \ll y$ is same as $x * 2^y$

$x \gg y$ is same as $x / 2^y$

অনেক ক্ষেত্রে এটি বেশ

সহায়ক, যেমন: $i * 256$;

এর পরিবর্তে লিখুন $b, i = i$

$\ll 8$;

মজার ব্যাপার হচ্ছে প্রায় সব কম্পাইলার নিজেই এধরনের পরিবর্তন করে নেয়। তাহলে আমাদের এটা শিখে লাভ কী? হ্যাঁ, লাভ আছে। ধরুন আরো জটিল কোনো এক্সপ্রেসন দেয়া হলো, (যেমন: $i = i \ll 8 + i \ll 4$ কে লেখা যায়, $i * 272$) কম্পাইলারের কিছু এ ব্যাপারটি বোঝার কথা নয়। যেমন, $x = 3 * n + 1$ জন্য লিখুন $x = n \ll 1 + n + 1$ অথবা $n + n + 1$ । আজকাল কনজিউমার ইলেক্ট্রনিক্স পণ্যের জন্য অপারেটিং সিস্টেম এবং সফটওয়্যারও তৈরি হচ্ছে, সুতরাং সেখানে আপনাকে মেমোরি এবং প্রসেসিং পাওয়ার দুয়েরই সীমাবদ্ধতার কথা মাথায় রাখতে হবে।

দ্রুততর লুপিং

লুপিং বা ইটারেশন একটি কমন ব্যাপার প্রায় যেকোনো প্রোগ্রামে। লুপিংয়ের ক্ষেত্রেও কিছু অপটিমাইজেশন টেকনিক প্রয়োগ করে প্রোগ্রামে দ্রুততা নিয়ে আসা যায়। আর অপটিমাইজেশনের উপকারিতাটা বেশি উপলব্ধ হয় যখন লুপের ভেতরে কিছু স্টেটমেন্টও থাকে।

লুপ পরিহার করা

কিছু লুপ আছে যা সরাসরি লুপ ব্যবহার না করেই লেখা যায় এমন ক্ষেত্রে লুপ পরিহার করুন। তাতে লুপ সেটআপ, ইনিশিয়ালাইজেশন, ইনক্রিমেন্ট এবং লুপ কাউন্টার চেক করা এসবের জন্য অযথা সময় অপচয় কমানো যায়। যেমন নিচের উদাহরণটি দেখুন—

```
for(int i = 0; i < 3; i++) array[i] = i;
```

মাত্র তিনটি এলিমেন্টের এ এ্যারেটি আপনি নিচের মতো করেই লিখুন।

```
array[0] = 0;
```

```
array[1] = 1;
```

```
array[2] = 2;
```

তবে মজার ব্যাপার হচ্ছে অনেক কম্পাইলার এ ব্যাপারটি প্রোগ্রামারদের চেয়েও স্মার্টভাবে করে নেয়। কারণ কম্পাইলার জানে, কতটা কোড এতে জেনারেট হবে এবং লুপ হলে ক্যাশ লাইন কতটা বড় হবে। কিন্তু তারপরও ভালো প্রোগ্রামারের গুণ হচ্ছে সবচেয়ে ভালো সল্যুশন তৈরি করা।

লুপের ভেতরের ক্যালকুলেশন পরিহারের চেষ্টা করুন—

```
for (int i = 0; i < nPixels; i++) {
    float b_value = v_ direction * 1_brightness * (1/v_distance);
    back_surface_bits[i] * b_value;
}
```

ওপরের লুপটি লক্ষ করুন, b_value-এর মানটির হিসেব যেমন বড় তেমনি এটি লুপের ভেতরে থাকাটাও জরুরি নয়। কারণ এটির সাথে লুপটির সরাসরি কোনো সম্পর্ক নেই।

এধরনের সমস্যা অন্যক্ষেত্রেও হতে পারে— লুপের ভেতরে বা অবজেক্ট কনস্ট্রাক্টরের ভেতরে অথবা ফাংশনের ইনিশিয়ালাইজেশন। সবসময়ই চিন্তাটা মাথায় রাখবেন, ‘যেটা করছি এটা কি সত্যিই দরকারি?’

Common Subexpression Elimination

হ্যাঁ, অথবা কোনো এক্সপ্রেশন রিপটেডভাবে ব্যবহার করবেন না।

```
if((dataStructPointer-> ExpensiveFunctionCall())< 10){
//some code here }
else if((data StructPointer->ExpensiveFunctionCall())> 30){
//some code here }
```

একে নিচের মতো করে লিখুন। কেননা বারবার কন্ডিশনটি ইভালুয়েশনে একই এক্সপ্রেশন আবারো ক্যালকুলেট করতে হচ্ছে যা বাঞ্ছনীয় নয়।

```
int temp = dataStructPointer->
ExpensiveFunctionCall();
if(temp<10){
//some code here }
else if(temp>30){
//some code here }
```

তবে এখানেও বলে নিচ্ছি কম্পাইলার সাধারণত এধরনের CSE একাই করে নেয়। সুতরাং ম্যানুয়ালি এটা করবেন তখন যখন কোনো ফাংশন কলের রেজাল্ট সেভ করার দরকার হবে, কেননা ফাংশন কলের কারণে অনেকসময় রেজিস্টার রিলোড হয় (বিশেষত রিকার্সিভ ফাংশনের বেলায়)।

এসেম্বলি ল্যাম্বুয়েজ ব্যবহার করুন

যদি আপনার কম্পাইলকৃত প্রোগ্রামের এসেম্বলি ভার্শনকে পর্যালোচনা করার অভ্যাস গড়ে তুলতে পারেন তাহলে নিজেই দেখতে পাবেন— প্রতি সেট সি++ স্টেটমেন্টের জন্য কতগুলি এসেম্বলি কোড জেনারেট হয়। এভাবে আপনি আরো জানতে পারবেন কোন ধরনের সি++ স্টেটমেন্টগুলো বেশি দ্রুততার সাথে রান হবে। যেমন ভিজুয়াল সি++ এ কোনো ভ্যারিয়েবলকে বাই নেম এক্সেস করলে দুইটি ইন্সট্রাকশনে রূপ নেয়, আর ঐ একই ভ্যালু যদি পয়েন্টার দিয়ে এক্সেস করার কোড লেখেন তা রূপ নেবে তিনটি ইন্সট্রাকশনে। তাই আমরা কথা নিশ্চয়ই বলতে পারি যে, ভ্যারিয়েবল নেম ব্যবহার করা পয়েন্টার ব্যবহারের চেয়ে দ্রুততর। তাই বলে আমি কিন্তু বলছি না যে পয়েন্টার ব্যবহার করা যাবে না। যাহোক, এভাবে যদি আপনি গবেষণা করতে থাকেন তাহলে আপনি একসময় এই জেনারেটেড এসেম্বলি কোডকেও অপটিমাইজ করতে পারবেন।

অর্থাৎ প্রোগ্রামের সময়নির্ভরশীল কোডগুলোকে কম্পাইল করে এসেম্বলিতে নিয়ে এবার তাকে আরো অপটিমাইজ করতে পারেন। নিচে আনসাইন্ড ডিভিশনের এক্স-৮৬ প্রসেসর ভিত্তিক এসেম্বলি কোডের উদাহরণ দেয়া হলো।

এ উদাহরণে একটি div (এবং একটি xor ও mov) কমানো যায় যদি আপনি ভাগফল ও ভাগশেষ দুটোই চান। অবশ্য, আরো ভালো কাজ করবে যদি div(stdlib.h এর) নামের ফাংশনটি ব্যবহার করি। উদাহরণটি দেখা যাক—

```
c = a/b; a, b, c এবং d হলো 32 bit integers
d = a%b;
এটি এসেম্বলিতে জেনারেট করলে নিচের কোডের
```

মতো হবে, এটি আপনি করতে পারেন যখন আপনার ভাগফল ও ভাগশেষ দুটোই দরকার।

```
xor edx, edx;// zero edx register
mov eax, a ;// move a to eax register
div b ;//divide eax by b
mov c,eax ;// move quotient to c
xor edx, edx ;// zero edx register
mov eax, a ;//move a to eax register
div b ; //divide eax by b
mov d, edx ;//move remainder to d
আর ওপরের কোডকে আরো অপটিমাইজ করলে
দাঁড়ায়—
xor edx, edx;// zero edx register
mov eax, a;// move a to eax register
div b;//divide eax by b
mov c,eax;// move quotient to c
mov d, edx;//move remainder to d
```

আপনি এধরনের কোড ইনলাইন এসেম্বলি হিসেবে আপনার প্রোগ্রামে সরাসরি ব্যবহার করতে পারেন। কিন্তু কম্পাইলকৃত সি++ এ লেখা প্রোগ্রাম-এর এসেম্বলি কোডকে যদি ব্যবচ্ছেদ করেন তবে এধরনের অদরকারি কোড বাদ দিতে পারেন। সেক্ষেত্রে আপনাকে পুরোটা নিজেই লিখতে হচ্ছে না।

স্পেশাল অপটিমাইজেশন

inline এবং register এ Keyword-গুলোর ব্যবহার।

inline

যদি ছোট ছোট এমন ফাংশন আপনার প্রোগ্রামে ব্যবহার করেন যা যখন-তখন কল করা হয়, তবে সেগুলো inline হিসেবে ডিক্লেয়ার করাই উচিত। আর এভাবে ডিক্লেয়ার করা হলে সে ফাংশনকে কল হিসেবে না ধরে তার মূল কোডকে এক্সিকিউটেবলে রাখে। এতে ফাংশন কল করার সময় অপচয় কমানো যায়। তবে স্মরণ রাখবেন, এ প্রক্রিয়া তখন কার্যকরী যখন আপনি অহরহ কল করা হবে এমন ছোট আকারের ফাংশনকে ইনলাইন করবেন।

আরো মজার ব্যাপার হচ্ছে, আপনি ইনলাইন করলেই সব ফাংশন ইনলাইনের মতো কাজ করবে না। ইনলাইন ফাংশনের নিচের বৈশিষ্ট্য থাকা চলবে না—

■ ভ্যালুরিটার্ন করা।

■ রিকার্সন।

■ আরেকটি ইনলাইন ফাংশনকে কলা করা।

■ ইটারেশন তথা লুপের ব্যবহার।

আর তা নাহলে কম্পাইলার ইনলাইন Keyword-টি উপেক্ষা করবে (এমনকি কোনো এরর বা ওয়ার্নিং নো দেখিয়েই)। আধুনিক কোনো কোনো কম্পাইলার ইনলাইন রিকার্সিভ ফাংশনে indepth (n=4) পর্যন্ত নিতে পারে, যা অনেক সময় সাশ্রয়ী। আবার কোনো কম্পাইলার হয়তো ওপরের সবগুলো নিয়ম উপেক্ষা করতে পারে, সেক্ষেত্রে তো দারুণ মজা।

register

কোনো ভ্যারিয়েবলকে সিপিইউ রেজিস্টারে স্টোর করতে হলে register টাইপ মডিফায়ারটি ব্যবহার করা হয়, এতে এক্সেস অপটিমাইজ হয়। তবে সিপিইউ রেজিস্টার সীমিত সংখ্যক হওয়ায়, ঠিক কোন কোন ভ্যারিয়েবলকে তাতে স্টোর করা উচিত সেই সিদ্ধান্তটি বুঝে শুনে নিতে হয়।

তবে for লুপের ক্ষেত্রে এর কাউন্টার ভ্যারিয়েবলকে রেজিস্টার স্টোর করা চলে। আর নেস্টেড লুপের ক্ষেত্রে তো সব ভ্যারিয়েবলকে রেজিস্টারে রাখার জায়গা পাবেন না, তাই সবচেয়ে ইনারমোস্ট লুপের ভ্যারিয়েবলকেই রেজিস্টারে রাখুন।

```
for(int i = 0; i<10; i++){
for(int j=0; j<10; j++){
for(register int k; k<10; k++) {
// Function body
}
}
}
```

এখানে দেখুন, i ১০ বার, j ১০০ বার এবং k ১০০০ বার ব্যবহৃত হচ্ছে সুতরাং সবচেয়ে ভেতরের লুপের চলককে রেজিস্টারে রাখলেই অনেকটা সাশ্রয় হচ্ছে। বলা বাহুল্য, আজকাল অনেক কম্পাইলার এই রেজিস্টার কীওয়ার্ড টি উপেক্ষা করে এবং নিজেই ঠিক করে কোন ভ্যারিয়েবলটি রেজিস্টার করলে বেস্ট পারফরম্যান্স পাওয়া যাবে।

Pointer Dereferencing

নিচের কোডটি দেখুন

```
for(int i = 0; i < BigNum; i++){
Inventory->StuffToSell->LowProfitStuff->
Count[i] = Value;
}
```

একে এভাবে লেখা যায়—

```
unsigned int *InventoryCount =
Inventory->StyffToSell-
>LowProfitStuff->Count;
for(int i = 0; i < BigNum; i++) {
```

Inventory Count[] = Value;
}

দ্বিতীয় কোডটি অত্যন্ত দ্রুততর হবে।

Temporary Variable পরিহার করুন

a = a + a1 + a2 + a3;

এর বদলে লেখা যায়—

a + = a1;

a + = a2

a + = a3;

দ্বিতীয় পদ্ধতিটিই দ্রুততর কেননা এতে কম্পাইলারের কোনো টেম্পোরারি অবজেক্ট তৈরি করার দরকার পড়ে না। যদিও কোড রিডাবিলিটি কমে যায়, যাক না:।

এভাবে, a = b + c; কে লেখা যায় a = b; a + = c;

এটাও টেম্পোরারি ভ্যারিয়েবল ব্যবহারের ব্যাপারটি দূর করে। আসলে প্রথম ক্ষেত্রে b এবং c দুয়কে একটি টেম্পোরারি ভ্যারিয়েবলে গ্যুড করা হয়, এরপর তাকে a-তে গ্যাসাইন করা হয়। অথচ দ্বিতীয় ক্ষেত্রে, টেম্পোরারি ভ্যারিয়েবল তৈরি হবে না।

কিন্তু এই অপটিমাইজেশন জটিলতর ভ্যারিয়েবল টাইপে (যথা— স্ট্রিং, যেখানে + অপারেটরটি একটি প্রিন্সিপাল করে টেম্পোরারি ভ্যারিয়েবলের বদলে) নাও কাজ করতে পারে।

String-এর তুলনা

String কে সি'তে গ্যারে হিসেবে ইম্প্লিমেন্ট করা হয়— আর এর কম্পারিজন অপারেশন একটি কমন ব্যাপার এবং তা বেশ ধীরগতিরও। সাধারণ পদ্ধতি হচ্ছে প্রতিটি এলিমেন্ট তুলনা করা এবং ডিফারেন্ট এলিমেন্ট পাওয়া মাত্র লুপ ব্রেক করা। যা খুবই ধীর গতিতে চলে যখন কোনো স্ট্রিংকে অন্য কতগুলো স্ট্রিং-এর সাথে তুলনা করতে হয় (এ ধরনের কাজ প্রায় প্রোগ্রামেই করতে হয়)—এর সমাধান হচ্ছে চেকসাম এর ব্যবহার। তবে, তার আগে অবশ্যই লেংথ ম্যাচ করে কি না দেখে নিতে হবে, কেননা ৯০ ভাগ ক্ষেত্রেই তা না মেলার সম্ভাবনা থাকে। চেকসাম হলো কোনো স্ট্রিং-এর সকল এলিমেন্টের (chars) আসকি ভ্যালুসমূহের যোগফল। নিচের উদাহরণ দেখুন—

```
unsigned char checksum = 0;
char str[20];
int len; // len= strlen(str);
//Get any string in str and the value of len,
//str may contain any string-text or binary
for(int i = 0; i < len; i++)
checksum+=str[i];
```

এভাবে সকল স্ট্রিংয়ের জন্য চেকসাম ক্যালকুলেট করুন। সময় নিঃসন্দেহে কিছুটা যাবে এই চেকসামের পেছনে কিন্তু বাঁচাবে তার অনেক গুণ। Char চেকসাম ২৫৬টি ভ্যালু রাখতে পারে, অর্থাৎ চেকসামে থাকবে ০-২৫৫ যে-কোনো মান। অর্থাৎ এক একটি স্ট্রিংয়ের জন্য গ্যাসাইন করা হবে ০-২৫৫ একটি ভ্যালু।

আমরা এক্ষেত্রে ক্লাস ব্যবহার করতে পারি।

```
class String {
char str [50];
unsigned char checksum;};
```

এই ক্লাসে একটি কনস্ট্রাক্টরও থাকা উচিত যার কাজ হবে যখন str-এ কোনো ভ্যালু ইনিশিয়ালাইজ অথবা আপডেট করা হবে তার জন্য চেকসাম তৈরি করা। একটি int len;ও থাকতে পারে প্রয়োজন হলে। আপনি এক্ষেত্রে কেবল প্রথম চারটি ক্যারেক্টারের জন্যই চেকসাম রাখতে পারেন (যা অনেক ক্ষেত্রে কাজে দেবে)। এটি ইকুয়ালিটির জন্য প্রয়োজ্য হলেও সর্টিংয়ের মতো ব্যাপারে কোনো কাজে আসবে না।

তাহলে আমাদের যখন কোনো দুটি স্ট্রিংয়ের মধ্যে তুলনার প্রয়োজন হবে; প্রথমেই তাদের চেকসাম মেলে কি না দেখব। যদি মেলে তবেই কেবল ক্যারেক্টার টু ক্যারেক্টার মাচিংয়ে আগাব।

Pre vs. Post Increment/Decrement

সবক্ষেত্রে প্রি-ইনক্রিমেন্ট এবং ডিক্রিমেন্ট ব্যবহার করুন, যেখানেই সম্ভব। কেননা, পোস্টফিক্স অপারেটর (i++) বর্তমান মানকে একটি টেম্পোরারি অবজেক্টে কপি করে, এরপর মান বৃদ্ধি করে, এরপর টেম্পোরারির মানকে রিটার্ন করে। আর প্রিফিক্স অপারেটর (++i) মানকে বৃদ্ধি করে এর রেফারেন্সকে রিটার্ন করে। আর এটা জেনে রাখবেন ইটারেটরের (যেমন লুপিং) মতো অবজেক্টের ক্ষেত্রে টেম্পোরারি কপি করাটা এক্সপেনসিভ।

Call by Value নাকি Call by Reference

আরগুমেন্টকে কোনো ফাংশনে বাই ভ্যালু (e.g.void f(Type x)] পাস করাটা বিল্ট-ইন টাইপের জন্য সাশ্রয়ী কিন্তু ক্লাসটাইপের জন্য তা এক্সপেনসিভ হবে, যদি এক্সেসের কপি কম্পট্রাক্টরটি নন-ট্রিভিয়াল হয়ে থাকে।

বাই গ্যার্ডেস পাসিং [e.g.void f (Type* x)] একটি লাইট-ওয়েট কলিং, কিন্তু ফাংশনটিও ডিফারেন্টভাবে কল করতে হয়, তাছাড়া পাসকৃত অবজেক্ট কলকৃত ফাংশনকর্তৃক বদলে দেবার ঝুঁকি থাকে। কোনো কনস্ট্যান্টকে বাই রেফারেন্স পাস করার বেলায়, পাসিং বাই ভ্যালুর নিরাপত্তা এবং পাসিং বাই গ্যার্ডেসের এফিসিয়েন্সি এ দুয়ের মিলিত সুবিধা পাওয়া যায়।

তবে আরগুমেন্ট ব্যবহারের বেলায় যেন কোনো অপ্রয়োজনীয় টেম্পোরারি অবজেক্ট তৈরি না হয়, যা অবশ্যই ঐ ফাংশনের প্যারামিটারের টাইপে কনভার্ট করতে হয়। অর্থাৎ, যদি কল করেন “void f (int const&x);” আর আরগুমেন্টে কোনো ফ্ল্যাটিং মান ব্যবহার করা হয় তাহলে কম্পাইলার যে কাজগুলো

করবে—

প্রথমে ইন্টিজার টাইপের একটি টেম্পোরারি ভ্যারিয়েবল তৈরি করবে তারপর ফ্ল্যাটিং ভ্যালুটি ইন্টিজারে কনভার্ট হবে ঐ টেম্পোরারি ভ্যারিয়েবলে গ্যাসাইনড হবে, পরিশেষে ফাংশনটি ঐ টেম্পোরারি ভ্যারিয়েবলের রেফারেন্সকে আরগুমেন্ট হিসেবে নেবে। আর এটা কোনোভাবেই কাম্য হওয়া উচিত নয়। একইরকম ব্যাপার ঘটবে যদি আমরা ফাংশনের আরগুমেন্টে কন্সট্যান্ট(ভ্যারিয়েবল-এর বদলে) দিয়ে তাকে কল করি।

Return করুন এভাবে...

রিটার্ন ভ্যালু অপটিমাইজেশন করে কম্পাইলারকে hint দিতে পারেন যে, টেম্পোরারি অবজেক্টস পরিহার করা যেতে পারে। ট্রিকটি হলো, অবজেক্ট রিটার্নের বদলে কম্পট্রাক্টর আরগুমেন্ট রিটার্ন করুন, নিচের মতো;

```
const Rational operator * (Rational
const & lhs, Rational const & rhs){
return Rational (lhs.numerator()*
rhs.numerator(),
lhs.denominator()*rhs.denominator ());
}
```

অসতর্কভাবেশত কোডিং-এ যে ভুলটি হতে পারে তা হলো, ক্যালকুলেশনের রেজাল্ট রাখার জন্য একটি লোকাল Rational ভ্যারিয়েবল তৈরি হতে পারে। কিন্তু এই hint ব্যবহারের ফলে, কম্পাইলার রিটার্ন ভ্যালুকে সরাসরি স্পেসিফাইড ভ্যারিয়েবলেই রাখবে।

Reallocate না করে Reuse করুন

মেমরি গ্যালোকেশন/ডি-গ্যালোকেশন অনেক সময় নেয়, তাই অলরেডি গ্যালোকটেড মেমরি চাংককেই রি-ইউজ করা উচিত (ডি-গ্যালোকটেড এবং আবার গ্যালোকটেড করার বদলে)। আর যদি অল্প অল্প পরিমাণ অবজেক্টের জন্য বারবার গ্যালোকটেড করেন, তার চেয়ে ভালো হলো একবারেই অনেক পরিমাণ মেমরি গ্যালোকটেড করে নেয়া এবং তার মধ্যেই আপনার ছোট অবজেক্টকে ম্যানেজ করা।

■ গাজী লেনিন

ghazilenin@hotmail.com

হুমায়ূন আহমেদ-এর

ন ত ন উ প ন্যা স

এই শুভ্র! এই

অন্যপ্রকাশ

৩৮-২ক, মান্নান মার্কেট (৩য় তলা),
বাংলাবাজার, ঢাকা-১১০০। ফোন : ৯১২৫৮০২

এখন
সারাদেশে
পাওয়া যাচ্ছে

