সহজ ভাষায় প্রোগ্রামিং সি/সি++

আহমদ ওয়াহিদ মাসুদ

উন্নতমানের প্রোগ্রাম বানানোর জন্য পয়েন্টারের ব্যবহার আবশ্যক। পয়েন্টারের ব্যবহারবিধি যেমন অনেক, তেমনই প্রকারভেদও অনেক। সব ধরনের পয়েন্টার সম্পর্কে ধারণা থাকলে ব্যবহারও সঠিকভাবে করা যায়। এ লেখায় বিভিন্ন ধরনের পয়েন্টার এবং এর অ্যাসোসিয়েটিভিটি নিয়ে আলোচনা করা হয়েছে।

অপারেটর অ্যাসোসিয়েটিভিটি এবং পয়েন্টার এক্সপ্রেশন

পয়েন্টারের সাথে কীভাবে *,++,-- ইত্যাদি ইউনারি অপারেটর ব্যবহার করা যায়, তা গত সংখ্যায় দেখানো হয়েছে। এসব ইউনারি অপারেটর পয়েন্টারের সাথে ব্যবহার করার সময় এদের মূল অ্যাসোসিয়েটিভিটি ও প্রিসিডেন্স মেনে চলে। অ্যাসোসিয়েটিভিটি ও প্রিসিডেন্স সম্পর্কে যাদের ভালো ধারণা নেই, তাদের জন্য সংক্ষিপ্ত বিবরণ দেয়া হলো। প্রোগ্রামে যে সবসময় সাধারণ এক্সপ্রেশন থাকবে তা নয়, জটিল এক্সপ্রেশনও থাকতে পারে। যেমন : x=5/4-6*2+81%9*(-2*-2)+(42+53)।

এটি একটি জটিল এক্সপ্রেশন। এখানে কোন অপারেটরের আগে কে কাজ করবে সে বিষয়টিকে বলা হয় অপারেটর প্রিসিডেন্স। আর একই ধরনের অপারেটরগুলো এক্সপ্রেশনের ডান দিক থেকে কাজ করবে না বাম দিক থেকে কাজ করবে, সে বিষয়টিকে বলা হয় অপারেটর অ্যাসোসিয়েটিভিটি। যেমন, আমরা জানি কোনো গাণিতিক এক্সপ্রেশনে যোগ/বিয়োগের থেকে গুণ/ভাগের কাজ আগে হয়। অর্থাৎ গুণ/ভাগের প্রিসিডেন্স যোগ/বিয়োগের থেকে আগে। এখানে শুধু কয়েকটি অপারেটরের অ্যাসোসিয়েটিভিটি দেখানো হয়েছে। কিন্তু সি-তে ব্যবহার হওয়া সব অপারেটরের জন্য নির্দিষ্ট অ্যাসোসিয়েটিভিটি এবং প্রিসিডেন্স আছে। অপারেটরের কাজ অনুযায়ী তাদের গ্রুপ করা হয়। এখানে *,++,-- হলো ইউনারি অপারেটর এবং এদের অ্যাসোসিয়েটিভিটি হলো ডান থেকে বাম দিকে। আর এদের সবার প্রিসিডেন্স সমান। তাই কোনো এক্সপ্রেশনে পয়েন্টার ভেরিয়েবলের সাথে দুটো অপারেটর ব্যবহার করা হলে এক্সপ্রেশনের মান বের করার জন্য সাধারণত অ্যাসোসিয়েটিভিটি ব্যবহার করা হয়। এ ধরনের অপারেটরের ব্যবহার ভালোভাবে বোঝার জন্য একটি উদাহরণ দেয়া হলো। প্রথমে ধরা যাক, ptr-কে নিচের মতো ডিক্লেয়ার করে মান নির্ধারণ করা হয়েছে।

```
int x,*ptr,a[5]={1,2,3,4,5};
ptr=a;
```

এখানে লক্ষণীয়, অ্যারে ডিক্লেয়ার করা এবং মান অ্যাসাইন করার পর তাকে পয়েন্ট করার সময় & অপারেটর ব্যবহার করা হয়নি। কারণ, কোনো অ্যারের নামই হলো এর প্রথম এলিমেন্টের অ্যাড্রেস। তাই সরাসরি ptr=a; লিখলে কোনো এরর দেখাবে না। সুতরাং ptr পয়েন্টারটি এখন অ্যারেটিকে পয়েন্ট করবে। অ্যারেটির অ্যাড্রেস যদি ৯০০১ থেকে শুরু হয়, তাহলে পয়েন্টারের ডাটা হিসেবে ৯০০১ স্টোর হবে। অর্থাৎ পয়েন্টারটি অ্যারেটির প্রথম এলিমেন্টটিকে পয়েন্ট করবে। আর অ্যারের প্রথম এলিমেন্টকে পয়েন্ট করা মানে অ্যারেটিকেই পয়েন্ট করা। এখন নিচের এক্সপ্রেশনটি দেখা যাক।

```
x=*ptr++;
```

যেহেতু * এবং ++ এর প্রিসিডেন্স একই। সুতরাং এ ক্ষেত্রে এক্সপ্রেশনের মান বের করার জন্য এদের অ্যাসোসিয়েটিভিটি ব্যবহার হবে। একটু আগেই বলা হয়েছে * এবং ++ এর অ্যাসোসিয়েটিভিটি হলো ডান দিক থেকে বাম দিক। তাই এখানে প্রথম ptr++ অংশটুকু কাজ করবে, যা প্রথমে পয়েন্টারের বর্তমান ডাটা রিটার্ন করবে এবং পরে তা ইনক্রিমেন্ট করবে। এরপর *(ptr++) অংশটুকু কাজ করবে, যা কিনা ptr এখন যাকে পয়েন্ট করছে তাকে রিটার্ন করবে। সুতরাং *ptr++ মানে হলো ptr, বর্তমানে যাকে পয়েন্ট করছে তাকে রিটার্ন করে পরে ইনক্রিমেন্ট করবে। তাই এ এক্সপ্রেশনটি কাজ করার পর x-এর মান ১ নির্ধারিত হবে, যা কিনা প্রথম এলিমেন্টের ডাটা। পরে ptr ইনক্রিমেন্ট হবে এবং তা দ্বিতীয় এলিমেন্টকে পয়েন্ট করবে। এখানে পয়েন্টারটি কীভাবে পরের এলিমেন্টকে পয়েন্ট করছে, তা ভালোমতো বোঝা দরকার। ptr পয়েন্টারের মাঝে অ্যারেটির প্রথম এলিমেন্টের অ্যাড্রেস ছিল। ptr ইনক্রিমেন্ট করার মানে হলো ওই অ্যাড্রেসকে ইনক্রিমেন্ট করা। আর প্রথম এলিমেন্টের অ্যাড্রেসকে ইনক্রিমেন্ট করলে যে অ্যাড্রেস পাওয়া যায় তা হলো দ্বিতীয় এলিমেন্টের অ্যাড্রেস। কারণ, মেমরিতে অ্যারের এলিমেন্টগুলো পরপর থাকে। এক্সপ্রেশনটি যদি হয় x=*++ptr; তাহলে ++ptr প্রথমে কাজ করবে এবং পরে *(++ptr) কাজ করবে। তাই *++ptr-এর মানে হলো পয়েন্টারের মান ইনক্রিমেন্ট করার পর সেটি যাকে পয়েন্ট করছে তাকে রিটার্ন করা। এ এক্সপ্রেশনটি এক্সিকিউট হওয়ার আগে পয়েন্টারটি দ্বিতীয় এলিমেন্টকে পয়েন্ট করছিল। এ এক্সপ্রেশনে আসার পর পয়েন্টারের মান প্রথমে ইনক্রিমেন্ট হবে, অর্থাৎ তা তৃতীয় এলিমেন্টকে পয়েন্ট করবে, তারপর সেই এলিমেন্টের মান রিটার্ন করবে। অর্থাৎ এখন x-এর নতুন মান হবে তৃতীয় এলিমেন্টের মান, যা কিনা ৩।

পয়েন্টার ক্যাস্টিং

সি-তে যদিও বিভিন্ন টাইপের ভেরিয়েবলের ব্যবস্থা রাখা হয়েছে, তবুও এমন অনেক অবস্থারই সৃষ্টি হতে পারে, যেখানে এক টাইপের ভেরিয়েবল অন্য টাইপ হিসেবে ব্যবহার করতে হতে পারে। যেমন, ইউজার অনেকগুলো ডাবল টাইপের ভেরিয়েবল নিয়ে কোনো কাজ করলেন। কিন্তু প্রোগ্রামের কোনো একটি জায়গায় এসে ওই ডাবল ভেরিয়েবলগুলোর মান প্রিন্ট করার প্রয়োজন হলো। এখন ইউজার চান মানগুলো দশমিক সংখ্যা হিসেবে প্রিন্ট না হয়ে পূর্ণসংখ্যা হিসেবে প্রিন্ট হবে। এ ক্ষেত্রে ইউজার চাইলে প্রিন্টের সময় ডাবল টাইপের ভেরিয়েবলগুলোকে ইন্টিজার টাইপ হিসেবে প্রিন্ট করতে পারেন। এতে ভেরিয়েবলের মানের কোনো পরিবর্তন হবে না। আবার ইউজার যদি চান একই ভেরিয়েবল কখনও পূর্ণসংখ্যা আবার কখনও দশমিক সংখ্যা ইউনপুট নেবে। সে ক্ষেত্রে কন্ডিশন দিয়ে একই ভেরিয়েবল এক সময় ইন্টিজার হিসেবে, আবার অন্য সময় ডাবল হিসেবে ব্যবহার করা যেতে পারে। এভাবে এক টাইপের ভেরিয়েবলকে অন্য টাইপ হিসেবে ব্যবহার করাকে ক্যাস্টিং করা বলে।

সি-তে পয়েন্টারকেও ক্যাস্ট করার ব্যবস্থা রাখা হয়েছে। ক্যাস্টিং নিয়ে নিচে একটি উদাহরণ দেয়া হলো।

```
double x=11.64,y=3.71;
int z;
z=((int)x)%((int)y);
```

এখানে % অপারেটরের অপারেন্ড হিসেবে x এবং y-কে ব্যবহারের আগে তাদের ইন্টিজার হিসেবে ক্যাস্ট করে নেয়া হয়েছে। কারণ z একটি ইন্টিজার টাইপের ভেরিয়েবল। তাই এর মাঝে ইন্টিজার ছাড়া অন্য কিছু স্টোর করা যাবে না। সুতরাং x এবং y-কে ইন্টিজারে ক্যাস্ট করে না নিলে তাদের দিয়ে কোনো এক্সপ্রেশনের মান z-তে স্টোর করা যাবে না।

এভাবে প্রোগ্রামে অনেক সময় পয়েন্টারকেও ক্যাস্ট করতে হয়। যাতে এক টাইপের পয়েন্টার দিয়ে অন্য টাইপের ডাটাকে পয়েন্ট করা যায় বা অন্য টাইপের ভেরিয়েবলের অ্যাড্রেস নিয়ে কাজ করা যায়। কারণ, বারবার পয়েন্টার ডিক্লেয়ার করা যেমন একদিকে সময়ের অপচয়, তেমনি মেমরিরও অপচয়। আর এছাড়া যত বেশি ভেরিয়েবল ডিক্লেয়ার করা হবে, কোডে এদের ম্যাপ করতে তত বেশি কষ্ট হবে। অর্থাৎ ইউজারের জন্য খুঁজে পেতে কষ্ট হবে কোডের কোন জায়গায় কোন ভেরিয়েবল বা পয়েন্টার ডিক্লেয়ার করা হয়েছে। এখন পয়েন্টারের ক্যাস্টিংয়ের একটি উদাহরণ দেয়া হলো।

```
double a=42.53;
char *ptr;
ptr=(char*)&a;
```

এখানে পয়েন্টারটি হলো ক্যারেক্টার টাইপের। তাই এ পয়েন্টারটি শুধু একটি ক্যারেক্টার টাইপের

▶

ভেরিয়েবলকে পয়েন্ট করতে পারবে বা কোনো ক্যারেক্টারের অ্যাড্রেস হোল্ড করতে পারবে। কিন্তু এখানে ক্যাস্টিংয়ের মাধ্যমে ডাবল টাইপের ভেরিয়েবলের অ্যাড্রেস পয়েন্টারে স্টোর করা হয়েছে। আমরা জানি (int)a মানে হলে a-এর ডাটাকে ইন্টিজারে রূপান্তর করে ব্যবহার করা। একইভাবে (char*)&a মানে হলো a-কে ইন্টিজারে রূপান্তর করে এর অ্যাড্রেস ব্যবহার করা। আর তাই শেষের লাইনে ptr-এ শুধু প্রথম বাইটের অ্যাড্রেসটিই পাওয়া যাবে, পুরো a-এর অ্যাড্রেস নয়। এখন,

```
printf(“%02x”,*ptr);
```

এ লাইনটির মাধ্যমে আউটপুটে প্রথম বাইটের ডাটা তথা ০০ দেখাবে। আবার ptr++; লেখা হলে পয়েন্টারটি পরের বাইটকে পয়েন্ট করবে। এখন প্রিন্ট করলে পরের বাইটের ডাটাগুলো প্রিন্ট হবে। এভাবে একবার করে ইনক্রিমেন্ট করে ক্যাস্টিংয়ের মাধ্যমে ডাটা ব্যবহার করা যেতে পারে।

নাল পয়েন্টার

বিভিন্ন বিল্ট-ইন ডাটা টাইপের জন্য সি-তে পয়েন্টার আছে। ডাটা টাইপের পয়েন্টারগুলো ছাড়াও কিছু পয়েন্টার আছে। এদের মধ্যে একটি হলো নাল পয়েন্টার। প্রোগ্রামিংয়ের ভাষায় নাল হলো কিছুই নেই। অর্থাৎ কোনো ডাটার অনুপস্থিতিকে নাল বলা যেতে পারে। তবে কোথাও ডাটা না থাকলেই কিন্তু সেটিকে নাল বলা যায় না। যেমন, প্রোগ্রামে যদি একটি লাইন প্রিন্ট করতে দেয়া হয় তাহলে বলা যাতে পারে লাইনটির শব্দগুলোর মাঝে যে ফাঁকা স্থান বা স্পেস আছে সেগুলো নাল। কারণ, সেখানে কোনো ডাটা নেই। কিন্তু স্পেসও একটি ডাটা। এখন পর্যন্ত মানুষ যত ধরনের সিম্বল ব্যবহার করে, হোক সেটি কোনো ভাষার অক্ষর বা ম্যাথ অথবা সায়েন্সের কোনো সিম্বল, সব ধরনের সিম্বলকে একটি কোড দেয়া হয়েছে, যাতে এদেরকে সঠিকভাবে প্রোগ্রামে ব্যবহার করা যায়। এ কোডকে ASCII কোড বলে। প্রোগ্রামিংয়ে ক্যারেক্টার নিয়ে আলোচনা করলে হোয়াইট স্পেস বলে একটি কথা অনেক ব্যবহার হয়। হোয়াইট স্পেস হলো সেসব ক্যারেক্টার, যাদেরকে দেখলে মনে হবে কোনো ডাটা নেই, কিন্তু আসলে আছে। যেমন, লাইনের মাঝে স্পেসও একটি ডাটা। স্পেসের আস্কি নাম্বার হলো ৩২ এবং এন্টার বা নিউ লাইনের আস্কি কোড হলো ১০। এ ধরনের স্পেস, নিউ লাইন ইত্যাদিকে হোয়াইট স্পেস বলা হয়। তাই এদের নাল ভেবে ভুল করা উচিত নয়। কারণ, নাল মানে হলো যেখানে কোনো ডাটা নেই। আর তাই নালের আস্কি কোড হলো ০। এমনকি প্রোগ্রামে কোনো লাইন প্রিন্ট করলে তার শেষে একটি নাল ক্যারেক্টার প্রোগ্রাম নিজেই বসিয়ে নেয়। কারণ, এতে করে প্রোগ্রাম পরে বোঝে কোথায় গিয়ে লাইনটি শেষ হয়েছে। লাইনের শেষে নাল না থাকলে প্রোগ্রাম কোনো লাইন প্রিন্ট করতে শুরু করত, কিন্তু তা আর শেষ হতো না। ইউজারের ইনপুট দেয়া লাইন প্রিন্ট করে পরে গারবেজ মান প্রিন্ট হতে থাকত। এ কারণেই প্রোগ্রাম নিজে থেকেই লাইনের শেষে একটি নাল ক্যারেক্টার বসিয়ে নেয়।

যে পয়েন্টার দিয়ে নাল ক্যারেক্টারকে পয়েন্ট করা যায় তাকে নাল পয়েন্টার বলা হয়। প্রোগ্রাম কোনো পয়েন্টার ভেরিয়েবলের জন্য অন্য ভেরিয়েবলের অ্যাড্রেস নির্ধারণ না করা পর্যন্ত তা আনইনিশিয়ালাইজড অবস্থায় থাকে অর্থাৎ ওই পয়েন্টারের ভেতরে কোনো ভেরিয়েবলের অ্যাড্রেস থাকে না। এ অবস্থায় ওই পয়েন্টার কোনো ভেরিয়েবলকে পয়েন্ট করে না, বরং কিছু গারবেজ ডাটাকে পয়েন্ট করে। একটি প্রোগ্রাম উদাহরণ হিসেবে দেয়া হলো।

```
int *ptr,x=42;
printf(“%x %02x\n”,ptr,*ptr);
ptr=&x;
printf(“%x %02x\n”,ptr,*ptr);
```

এখানে পয়েন্টারকে ডিক্লেয়ার করার সময় ইনিশিয়ালাইজড করা হয়নি। তাই পয়েন্টারের জন্য কোনো অ্যাড্রেস নির্ধারিত হবে না। অর্থাৎ এতে কিছু গারভেজ মান থাকবে। এখন প্রথম প্রিন্টে প্রোগ্রাম চলার সময় পয়েন্টারের জন্য যে সেলগুলো ব্যবহার করা হবে সেই সেলে যে সংখ্যা পাওয়া যাবে *ptr-এর মাধ্যমে সেই সংখ্যা অনুযায়ী অ্যাড্রেসের ডাটা দেখানো হবে। এখানে আউটপুট হিসেবে *ptr-এর মাধ্যমে 2FF পাওয়া যাবে। কারণ, পয়েন্টারের জন্য যে সেলগুলো ব্যবহার হয়েছে সেখানে 35C আছে এবং 35C অ্যাড্রেসে 2FF আছে। তবে একেক কমপিউটারে এ আউটপুট একেক ধরনের হতে পারে। কারণ, এটি একটি র‌্যান্ডম অ্যাড্রেস। তবে এটি কোনো বড় ইস্যু নয়। কারণ, সব কমপিউটারেই কাজটি একইভাবে হবে। কিন্তু দ্বিতীয় প্রিন্টের আগে যেহেতু পয়েন্টারের জন্য x-এর অ্যাড্রেস নির্ধারণ করা হয়েছে। তাই এ প্রিন্টের মাধ্যমে x-এর অ্যাড্রেস পাওয়া যাবে এবং *ptr দিয়ে x-এর মান দেখা যাবে। এভাবে পয়েন্টারকে যতক্ষণ পর্যন্ত না ইনিশিয়ালাইজড করা হবে, ততক্ষণ পর্যন্ত তা গারবেজ ডাটাকে পয়েন্ট করে থাকবে। কিন্তু প্রোগ্রামে যদি এ ধরনের পরিস্থিতির সৃষ্টি হয় যে কোনো পয়েন্টারকে যতক্ষণ না পর্যন্ত ইনিশিয়ালাইজড করা হবে সেটি কোনো কোনো গারবেজ ডাটাকেও পয়েন্ট করতে পারবে না, সে ক্ষেত্রে নাল পয়েন্টার ব্যবহার করা যেতে পারে। যেমন–

```
char *ptr=0;
char *ptr=null;
```

এখানে ptr-কে সাধারণ নাল পয়েন্টার বলা হয়। উল্লেখ্য, আস্কি স্ট্যান্ডার্ড অনুযায়ী stdio, stdlib অথবা string লাইব্রেরিতে নালের নিচের যেকোনো একভাবে ডিফাইন করা থাকতে পারে।

```
#define NULL 0
#define NULL (void*)0
```

এভাবে কোনো পয়েন্টারকে নাল হিসেবে ডিক্লেয়ার করার ফলে তা যেমন কোনো ভেরিয়েবলকে পয়েন্ট করবে না, তেমনি কোনো গারবেজ ডাটাকেও পয়েন্ট করবে না। ইউজারের উচিত শুরুতে পয়েন্টারকে নাল হিসেবে ডিক্লেয়ার করে নেয়া। কারণ বড় প্রোগ্রামের ক্ষেত্রে অনেক সময় কোনো পয়েন্টার দিয়ে যদি কোনো ভেরিয়েবলকে পয়েন্ট করতে ইউজার ভুলে যান, তাহলে প্রোগ্রাম অপ্রত্যাশিত ফলাফল দেবে এবং কেনো দেবে, দিচ্ছে তা ইউজারের জন্য বের করা অনেক কষ্টসাধ্য হয়ে যাবে। বড় প্রোগ্রামে অসংখ্য কোড থাকে এবং তা থেকে ভুল বের করা খুবই কঠিন একটি কাজ। আবার বড় প্রোগ্রাম লেখার সময় ভুল হওয়াটা একেবারেই স্বাভাবিক। কিন্তু পয়েন্টারকে যদি শুরুতে নাল হিসেবে ডিক্লেয়ার করা হয়, তাহলে প্রোগ্রাম অপ্রত্যাশিত ফলাফল বা গারবেজ মান প্রিন্ট না করে ০ (শূন্য) প্রিন্ট করবে এবং এটি দেখে ইউজার সহজেই বুঝতে পারবেন কোথায় ভুল হয়েছে।

বিভিন্ন ধরনের পয়েন্টারের ব্যবহার প্রোগ্রামকে অনেক সহজ এবং উন্নতমানের করে তোলে। তাই বিভিন্ন ধরনের পয়েন্টারের ব্যবহার সম্পর্কে জানা প্রয়োজন কজ

ফিডব্যাক : wahid_cseaust@yahoo.com