সহজ ভাষায় প্রোগ্রামিং সি/সি++

আহমদ ওয়াহিদ মাসুদ

আধুনিক সব ল্যাঙ্গুয়েজ সি-কে অনুকরণ করেই বানানো হয়েছে। কারণ সি-তে একইসাথে যেমন একেবারে লো লেভেলে, অর্থাৎ মেমরির অ্যাড্রেস লেভেলে কাজ করা যায়, তেমনি অনেক হাই লেভেল কোডিংয়ের সুবিধাও আছে। ব্যবহারকারীদের কোড করার সুবিধার্থে এবং কষ্ট কমানোর জন্য সি-তে অনেক হাই লেভেল ফিচার আছে। সি ল্যাঙ্গুয়েজের অনুকরণে অন্যান্য আধুনিক ল্যাঙ্গুয়েজেও এসব সুবিধা দেয়া হয়। সি ল্যাঙ্গুয়েজের একটি অন্যতম ফিচার হলো কাস্টম ডাটা টাইপ ব্যবহার করা। গত পর্বে কাস্টম ডাটা টাইপ হিসেবে স্ট্রাকচার নিয়ে বেসিক আলোচনা করা হয়েছে। এ পর্বে দেখানো হয়েছে কীভাবে স্ট্রাকচারের সাথে অ্যারে, পয়েন্টার, ফাংশন, নেস্টেড স্ট্রাকচার ইত্যাদি ব্যবহার করা যায়।

নেস্টেড স্ট্রাকচার

প্রোগ্রামে যেভাবে নেস্টেড লুপ ব্যবহার করা যায়, তেমনি প্রোগ্রামারের সুবিধার্থে নেস্টেড স্ট্রাকচার ব্যবহার করারও সুযোগ আছে। অর্থাৎ ব্যবহারকারী ইচ্ছে করলে একটি স্ট্রাকচারের মেম্বার হিসেবে আরেকটি স্ট্রাকচার ব্যবহার করতে পারবেন। যেমন :

```
struct student
{
        char*     name;
        int                 id;
        struct term
        {
                double    gpa;
                int       courseID;
        }course;
}data;
```

এখানে স্টুডেন্ট নামের স্ট্রাকচারের ভেতরে তিনটি মেম্বার ডিক্লেয়ার করা হয়েছে, যার মাঝে শেষের মেম্বারটি নিজেই একটি স্ট্রাকচার। লক্ষ করলে দেখা যাবে, এখানে স্ট্রাকচার ডিক্লেয়ার করার সাথে সাথেই তাদের ভেরিয়েবল ডিক্লেয়ার করা হয়েছে। যেমন : ভেতরের মেম্বার স্ট্রাকচারের ভেরিয়েবলটির নাম কোর্স এবং বাইরের অর্থাৎ মেইন স্ট্রাকচারের ভেরিয়েবলের নাম ডাটা। সুতরাং এখন এই নেস্টেড স্ট্রাকচারের ভেরিয়েবলগুলোর মান নিচের মতো করে নির্ধারণ করা যাবে :

```
    data.name=”wahid”;
    data.id=53;
    data.course.gpa=3.5;
    data.course. courseID=33;
```

এখানে নাম হচ্ছে ডাটার একটি মেম্বার, আইডি ও শুধু ডাটার মেম্বার, কিন্তু জিপিএ হলো কোর্সের মেম্বার আর কোর্স হলো ডাটার মেম্বার। তাই এক্ষেত্রে দুইবার মেম্বার অপারেটর ব্যবহার করতে হয়েছে। কোর্স আইডির মানও একইভাবে নির্ধারণ করা হয়েছে। অর্থাৎ শুধু মেম্বার অপারেটর দিয়েই নেস্টেড স্ট্রাকচারের সব মেম্বারের মান নির্ধারণ করা যাবে। একইভাবে ব্যবহারকারী যদি এই ভেরিয়েবলগুলো ব্যবহার করে অন্য কোনো কাজ যেমন : ইনপুট বা প্রিন্ট ইত্যাদি করতে চাইলে এই মেম্বার অপারেটর ব্যবহার করতে হবে।

তবে এভাবে নেস্টেড স্ট্রাকচার ব্যবহার করতে ব্যবহারকারীর অনেক সময় সমস্যা হতে পারে। স্ট্রাকচারের আকার যদি অনেক বড় হয়ে যায় আর একটি মেইন স্ট্রাকচারের ভেতরে যদি অনেকগুলো নেস্টেড স্ট্রাকচার থাকে, তাহলে সেখানে ভুল হওয়ার সম্ভাবনা অনেক বেশি। সে ক্ষেত্রে উপরের একই স্ট্রাকচারকে নিচের মতো করে ডিক্লেয়ার করা যেতে পারে।

```
struct term
{
        double    gpa;
        int       courseID;
};
struct student
{
        char*     name;
        int                 id;
        struct term course;
}data;
```

অর্থাৎ ব্যবহারকারী চাইলে ভেতরের সব স্ট্রাকচারকে আগে আলাদাভাবে ডিক্লেয়ার করে পরে মূল স্ট্রাকচারকে ডিক্লেয়ার করতে পারেন। সে ক্ষেত্রে মূল স্ট্রাকচারের ভেতরে উপরের উদাহরণের মতো শুধু ওই নেস্টেড স্ট্রাকচারগুলোর ভেরিয়েবল লিখে দিলেই হবে। তবে এ ক্ষেত্রে কোড কিছু বেশি লিখতে হয়। তাই প্রোগ্রাম যদি বেশি বড় না হয়, তাহলে সরাসরি ডিক্লেয়ার করাই ভালো। কারণ, প্রোগ্রাম বড় হলে আলাদা কথা, কিন্তু ছোট প্রোগ্রামে অযথা বড় কোড লিখলে একইসাথে কোডারের কষ্ট যেমনি বাড়ে, তেমনি প্রোগ্রামের কম্পাইল টাইম ও রান টাইম দুটোই বেড়ে যায়। অর্থাৎ প্রোগ্রাম কিছুটা হলেও বেশি রিসোর্স টানে।

স্ট্রাকচার ভেরিয়েবলের অ্যারে

সি-তে একসাথে অনেকগুলো স্ট্রাকচার ভেরিয়েবল ব্যবহার করার সুযোগ রয়েছে। তাই প্রোগ্রামে যেকোনো ভেরিয়েবলের মতো স্ট্রাকচার ভেরিয়েবলেরও অ্যারে ডিক্লেয়ার সম্ভব। যেমন : `struct student data[3];`

এখানে ডাটা নামের একটি স্ট্রাকচার অ্যারে ডিক্লেয়ার করা হলো। তবে ব্যবহারকারী চাইলে অন্যভাবেও অ্যারেটি ডিক্লেয়ার করতে পারেন। যেমন :

```
struct student data[]={
{“rahim”,01,3.5,101},
{“karim”,02,3.6,202},
{“jamal”,03,3.7,303}
}
```

এভাবে ব্যবহারকারী চাইলে সরাসরি অ্যারের মান ডিক্লেয়ার করার সময় অ্যাসাইন করে দিতে পারেন। যেমন : এখানে ডাটা নামের অ্যারে ভেরিয়েবলের তিনটি এলিমেন্ট ডিক্লেয়ার করা হয়েছে। প্রতিটি এলিমেন্টের ভেতরে মূল স্ট্রাকচার ভেরিয়েবলের সিকোয়েন্স অনুযায়ী মান অ্যাসাইন করা হয়েছে। এই সিকোয়েন্স যদি ঠিক না থাকে, তাহলে এরর দেখাতে পারে, অথবা এরর না দেখালেও ফলাফল ভুল হতে পারে অথবা গারবেজ ভ্যালু চলে আসতে পারে। তাই এভাবে অ্যারে ডিক্লেয়ার করার সময় এই সিকোয়েন্সের দিকেই বেশি খেয়াল রাখতে হবে। আরেকটি গুরুত্বপূর্ণ বিষয় হলো, অ্যারে ভেরিয়েবলের মান উপরের মতো করে অ্যাসাইন করার সময় প্রতিটি এলিমেন্টের পরে একটি করে কমা দিতে হবে, কিন্তু শেষ এলিমেন্টের পরে কোনো কমা দেয়া যাবে না। দিলে এরর দেখাবে।

স্ট্রাকচারের ভেরিয়েবলকেই শুধু অ্যারে হিসেবে ডিক্লেয়ার করা যায় না, এর মেম্বারকেও অ্যারে হিসেবে ব্যবহার করা যায়। যেমন :

```
struct student
{
char                    name[4];
                        }data;
```

এখানে ডাটা ভেরিয়েবলের মেম্বার হিসেবে একটি ক্যারেক্টার অ্যারে ব্যবহার করা হয়েছে। তাই একে ব্যবহার করতে হলে নিচের মতো কোড লিখতে হবে :

```
    data.name[0]=’A’;
```

আর এ ক্ষেত্রে যদি ডাটা ভেরিয়েবলের অ্যারে থাকত, তাহলে এই কোডটি লিখতে হবে :

```
    data[0].name[0]=’A’;
```

একইভাবে এখানে যদি কোনো নেস্টেড স্ট্রাকচারের অ্যারে থাকত, তাহলে তাকেও সাধারণ অ্যারের নিয়মানুসারে অ্যাক্সেস করা যেত।

স্ট্রাকচার ও পয়েন্টার

সি ল্যাঙ্গুয়েজে স্ট্রাকচারের মাঝে পয়েন্টারের ব্যবহারের সুবিধাও রয়েছে। প্রোগ্রামে যে নিয়মে সাধারণ পয়েন্টার ব্যবহার করা হয়, সেই একই নিয়মে স্ট্রাকচারের মাঝেও পয়েন্টার ব্যবহার করা যাবে। যেমন :

```
struct student
{
        char*     name;
}*data,info;
```

এখানে প্রথমে স্ট্রাকচারের ভেতরে ক্যারেক্টার পয়েন্টার ভেরিয়েবল নেম ডিক্লেয়ার করা হয়েছে। পরে ওই স্ট্রাকচারের একটি পয়েন্টার ভেরিয়েবল ডাটা ডিক্লেয়ার করা হয়েছে। অর্থাৎ স্ট্রাকচারের যেকোনো জায়গায় ব্যবহারকারী চাইলে পয়েন্টার ডিক্লেয়ার করতে পারেন।

স্ট্রাকচারের ক্ষেত্রে পয়েন্টার নিয়ে কাজ করলে তাকে অ্যাক্সেস করার নিয়মে একটু ভিন্ন রকম হয়। যেমন : কোনো পয়েন্টেড ভেরিয়েবলের মেম্বার নিয়ে কাজ করার সময় মেম্বার অপারেটর ব্যবহার না করে অ্যারো অপারেটর নিয়ে কাজ করতে হয়। অর্থাৎ সে ক্ষেত্রে মেম্বার ভেরিয়েবলের সিনটেক্স হবে :

```
    pointer_variable_name →
member_name;
```

যেমন উপরের পয়েন্টার স্ট্রাকাচারের একটি পয়েন্টার মেম্বারকে যদি প্রিন্ট করতে হয়,

```
    data=&info;
```

▶

```
printf("%s",data→name);
```

লক্ষণীয়, ডাটা একটি পয়েন্টার ভেরিয়েবল, তাই এর নিজের কোনো মেম্বার নেই। এটি যদি অন্য কাউকে পয়েন্ট করে, তাহলে এর মেম্বারকে অ্যাক্সেস করতে পারবে। আবার ডাটা পয়েন্টারটি সবাইকে পয়েন্ট করতে পারবে না, যাকে পয়েন্ট করবে তার স্ট্রাকচার টাইপ এবং ডাটার স্ট্রাকচার টাইপ একই হতে হবে। এখানে ডাটা এবং ইনফো, এ দুইটি ভেরিয়েবল একই স্ট্রাকচার থেকে ডিক্লেয়ার করা হয়েছে। পার্থক্য হলো– ডাটা একটি পয়েন্টার ভেরিয়েবল। এখন ইনফো যেহেতু পয়েন্টার নয়, তাই তার মেম্বার থাকা সম্ভব। আর ডাটা যেহেতু পয়েন্টার, তাই এর মেম্বার থাকা সম্ভব নয়। তাই আগে যদি ডাটা ইনফোকে পয়েন্ট করে, তাহলে পরে ডাটার মাধ্যমে ইনফোর মেম্বারকে অ্যাক্সেস করা যাবে। উপরের উদাহরণে আসলে তাই করা হয়েছে। প্রথমে ডাটা ইনফোকে পয়েন্ট করেছে। এরপর ডাটাকে দিয়ে ইনফোর মেম্বারকে প্রিন্ট করা হয়েছে। এখানে অনেক সময় একটি সাধারণ ভুল হতে দেখা যায়। প্রোগ্রামাররা মাঝেমধ্যে পয়েন্টার দিয়ে ভুলে অন্য ভেরিয়েবলকে পয়েন্ট না করেই তার মেম্বারকে অ্যাক্সেস করার চেষ্টা করেন।

স্ট্রাকচার ভেরিয়েবল ও স্ট্রাকচার পয়েন্টারের পার্থক্য

একটি পয়েন্টারের সাথে যেকোনো ভেরিয়েবলেরই পার্থক্য থাকে। স্ট্রাকচারের ক্ষেত্রেও এর বিকল্প নেই। আগে সাধারণ ভেরিয়েবল এবং পয়েন্টার নিয়ে কাজ করার সময় দেখানো হয়েছে পয়েন্টারকে সাধারণ ভেরিয়েবলের মতো ব্যবহার করা যায় না, এর কিছু ভিন্ন ধরনের নিয়ম রয়েছে। আবার বিভিন্ন ধরনের পয়েন্টার মেমরিতে কীভাবে অবস্থান করে, তাও দেখানো হয়েছে। স্ট্রাকচারের ক্ষেত্রেও একইভাবেই পয়েন্টার মেমরিতে অবস্থান করে। পার্থক্য হলো এ ক্ষেত্রে ওই পয়েন্টারকে অ্যাক্সেস করতে হলে স্ট্রাকচারের মাধ্যমে করতে হবে।

একটি স্ট্রাকচারের পয়েন্টারের জন্য সবসময় ২ থেকে ৪ বাইট জায়গা নির্ধারিত হয়। পয়েন্টারটি যদি মেম্বার পয়েন্টার হয়, তাহলেও একই নিয়ম প্রযোজ্য। কিন্তু একটি স্ট্রাকচারের বেলায় ওই স্ট্রাকচারের মেম্বারগুলো নির্ভর করে স্ট্রাকচার ভেরিয়েবলটি মেমরিতে কতটুকু জায়গা দখল করবে। স্ট্রাকচার পয়েন্টার যেহেতু আসলে একটি পয়েন্টার মাত্র, তাই এর নিজের কোনো মেম্বার থাকা সম্ভব নয়। কিন্তু এর মাধ্যমে অন্য কারও মেম্বারকে অ্যাক্সেস করা সম্ভব। কোনো স্ট্রাকচার পয়েন্টার দিয়ে যদি কোনো স্ট্রাকচারকে আগে পয়েন্ট করা হয়, তাহলেই শুধু পরে ওই পয়েন্ট করা স্ট্রাকচার ভেরিয়েবলের মেম্বারকে অ্যাক্সেস করা যাবে। আর স্ট্রাকচারের মেম্বারকে অ্যাক্সেস করতে হলে মেম্বার অপারেটর ব্যবহার করতে হয়, কিন্তু যখন পয়েন্টার দিয়ে মেম্বারকে অ্যাক্সেস করার দরকার হবে, তখন অ্যারো অপারেটর ব্যবহার করতে হবে। একটি স্ট্রাকচার ভেরিয়েবলের মাঝে বিভিন্ন ধরনের ডাটা থাকে, কিন্তু একটি পয়েন্টারের মাঝে সবসময় শুধু অ্যাড্রেসই থাকবে। এ কারণেই পয়েন্টার শুধু ২ বাইট অথবা কম্পাইলার বিশেষে ৪ বাইট জায়গা নেয়। কারণ মেমরির অ্যাড্রেস ধারণ করতে সাধারণত ৪ বাইটের বেশি দরকার হয় না।

স্ট্রাকচার ও ফাংশন

ফাংশনের প্যারামিটার, রিটার্ন ডাটা টাইপ কিংবা লোকাল ভেরিয়েবল ইত্যাদি বিভিন্ন প্রয়োজনে স্ট্রাকচার ব্যবহার করা যেতে পারে। ফাংশনের প্যারামিটার হিসেবে সম্পূর্ণ স্ট্রাকচারকে বা নির্দিষ্ট কোনো মেম্বারকে কিংবা স্ট্রাকচারের অ্যাড্রেসকেও পাঠানো যায়।

প্যারামিটার হিসেবে স্ট্রাকচার ভেরিয়েবল

প্যারামিটার হিসেবে স্ট্রাকচার ভেরিয়েবল ব্যবহার করতে নিচের মতো করে ডেফিনিশন লিখতে হবে :

```
return_type function_name (struct tag variable_name) {....}
```

এবং ফাংশনকে এভাবে কল করতে হবে :

```
function_name(tag type variable_name);
```

উদাহরণ হিসেবে ছোট একটি প্রোগ্রাম দেয়া যায় :

```
struct person
{
        char*     name;
        int                 age;
}
showPersonInfo(struct person p1)
{....}
struct person iqbal;
showPersonInfo(iqbal);
```

এখানে প্রথমে একটি স্ট্রাকচার ডিফাইন করা হয়েছে। এরপর একটি ফাংশনের ডেফিনিশন লেখা হয়েছে। এরপর ওই স্ট্রাকচারের ইকবাল নামে একটি ভেরিয়েবল তৈরি করে পরের লাইনে ইকবাল ভেরিয়েবলটিকে প্যারামিটার হিসেবে ওই ফাংশনে পাঠিয়ে দেয়া হয়েছে।

প্যারামিটার হিসেবে স্ট্রাকচার পয়েন্টার

এটিও আগের মতো অনেকটা একইভাবে লিখতে হয়। শুধু পার্থক্য হলো, ফাংশন ডেফিনিশন লেখার সময় প্যারামিটারের জায়গায় স্ট্রাকচার টাইপের পরে একটি অ্যাস্টেরিক সাইন (*) দিতে হয়। আর ফাংশনকে কল করার সময় প্যারামিটার হিসেবে অ্যাড্রেস পাঠানোর জন্য অপারেটর ব্যবহার করতে হয়। যেমন :

```
showPersonInfo(struct person* p1)
{....}
showPersonInfo(&iqbal);
```

এভাবে ব্যবহারকারী নিজের ইচ্ছেমতো ফাংশনের প্যারামিটার হিসেবে স্ট্রাকচার ব্যবহার করতে পারবেন। শুধু তাই নয়, ইচ্ছে করলে স্ট্রাকচারকেও রিটার্ন ভ্যালু হিসেবে ব্যবহার করা যাবে। এ রিটার্নের সময় ব্যবহারকারী চাইলে স্ট্রাকচার পয়েন্টার অথবা সাধারণ স্ট্রাকচার ভেরিয়েবল দুটোই রিটার্ন করতে পারেন।

বিভিন্ন ধরনের কাস্টম ডাটা টাইপের মাঝে স্ট্রাকচারের ব্যবহারই সবচেয়ে বেশি লক্ষ করা যায়। স্ট্রাকচারের ব্যবহার সি ল্যাঙ্গুয়েজের একটি অন্যতম ফিচার। এটি ব্যবহারের মাধ্যমে ব্যবহারকারীর কোড করা অনেক সহজ হয়ে যায়। সাথে প্রোগ্রামের গুণগত মানও অনেক বাড়ে। এছাড়া বড় ধরনের প্রোগ্রাম লেখার সময় এমন অনেক পরিস্থিতির সৃষ্টি হতে পারে, যেখানে স্ট্রাকচার অথবা অন্যান্য কাস্টম ডাটা ব্যবহার না করে কোনো উপায় নেই। তাই স্ট্রাকচারের ব্যবহার ভালোভাবে জানা একজন প্রোগ্রামারের জন্য জরুরি কজ

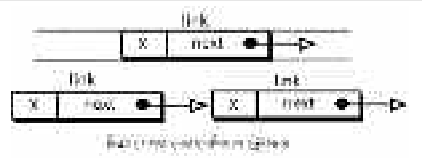
ফিডব্যাক : wahid_cseaust@yahoo.com

সেলফ রেফারেন্সিয়াল স্ট্রাকচার

সি-তে কোনো স্ট্রাকচারের মেম্বার হিসেবে সেই স্ট্রাকচার টাইপের পয়েন্টার ডিক্লেয়ার করা যায় এবং এ ধরনের স্ট্রাকচারকে সেলফ রেফারেন্সিয়াল স্ট্রাকচার বলে। বিষয়টি অনেকটা ফাংশনের রিকার্সনের মতো। যেমন :

```
struct node
{
        int                 x;
        struct node*        next;
}link;
```

এই স্ট্রাকচারে দুইটি মেম্বার আছে। একটি ইন্টিজার টাইপের এক্স নামের ভেরিয়েবল, যার কাজ হলো একটি সংখ্যা ধারণ করা। অন্যটি হলো struct node টাইপের পয়েন্টার। ধরা যাক, প্রোগ্রামে লিঙ্ক নামে ওই স্ট্রাকচারের একটি ভেরিয়েবল তৈরি করা হলো। ফলে চিত্র-১-এর প্রথম ছবির মতো করে তা মেমরিতে জায়গা দখল করবে। এখানে প্রথম ভেরিয়েবলের কাজ ডাটা রাখা আর দ্বিতীয় ভেরিয়েবলের কাজ struct node টাইপের কোনো স্ট্রাকচার ভেরিয়েবলের অ্যাড্রেস রাখা। ধরা যাক, প্রোগ্রামে কয়েকটি এরকম পয়েন্টার ডিক্লেয়ার করা হলো। তাহলে সেগুলো চিত্র-১-এর দ্বিতীয় ছবির মতো একটি আরেকটিকে পয়েন্ট করে থাকতে পারবে। ফলে একটি পয়েন্টারের চেইন তৈরি হবে। প্রোগ্রামিংয়ে



এ ধরনের চেইনের বেশ জনপ্রিয়তা আছে। সাধারণত ডায়নামিক প্রোগ্রামিংয়ে এটি অনেক ব্যবহার করা হয়। ডায়নামিক প্রোগ্রামিংয়ের সুবিধা হলো মেমরিতে জায়গা অনেক কম প্রয়োজন হয়। কারণ সাধারণ প্রোগ্রামিংয়ে অনেকগুলো ভেরিয়েবল একসাথে ডিক্লেয়ার করা হয়, কিন্তু তার কিছু কিছু ব্যবহার করা হয়, আবার কিছু কিছু ফাঁকা থেকে যায়। কিন্তু মেমরিতে ওই ফাঁকা ভেরিয়েবলগুলো ঠিকই জায়গা দখল করে রাখে। এখানেই ডায়নামিক প্রোগ্রামিংয়ের সাথে সাধারণ প্রোগ্রামিংয়ের পার্থক্য। কারণ ডায়নামিক প্রোগ্রামিংয়ে আগে থেকে কোনো ভেরিয়েবল মেমরিতে জায়গা দখল করে না। যখন ব্যবহারকারী একটি ভেরিয়েবল নিয়ে কাজ করবেন, তখন তা স্বয়ংক্রিয়ভাবে মেমরিতে নিজের জন্য জায়গা দখল করে নেবে। তাই এ ক্ষেত্রে মেমরিতে ফাঁকা ভেরিয়েবল থাকার কোনো সম্ভাবনা থাকে না। এ ধরনের চেইনের ব্যবহার যেহেতু ডায়নামিক প্রোগ্রামিংয়ের ভেতরে পড়ে, তাই তা পরবর্তী সংখ্যায় দেখানো হবে।