Gift for Programming Students

Ameen Ahmad
Virtual university

فہرستِ عنوانات

	عرضِ مصنف- کچھ اس کتاب کے بارے میں
	عرضِ ناشر......یا......
باب نمبر1	کمپیوٹر پروگرامنگ کا تعارف
باب نمبر2	پروگرامنگ لینگویجز کی تاریخ اور ++C کی ابتداء
باب نمبر3	بنیادی اصطلاحات اور ++C کا پہلا پروگرام
باب نمبر4	ویری ایبلز اور ڈیٹا ٹائپس
باب نمبر5	Format Specifiers اور Operators
باب نمبر6	Decision Statements
باب نمبر7	LOOPS
باب نمبر8	فنکشنز(Functions)
باب نمبر9	ایڈوانسڈ فیچرز(Advanced Features)
باب نمبر10	اسٹرکچرز(Structures)
باب نمبر11	یونین(Union)
باب نمبر12	ایرے(Array)
باب نمبر13	پوائنٹرز(Pointers)
باب نمبر14	گرافکس(Graphics)
باب نمبر15	فائلز(Files)
باب نمبر16	آبجیکٹ اورینٹڈ پروگرامنگ(OOP) کا تعارف
باب نمبر17	Encapsulation, Constructors and Destructors
باب نمبر18	اسٹرنگز(Strings)
باب نمبر19	اسٹوریج کلاسز(Storage Classes)
باب نمبر20	کلاسز اور آبجیکٹس
باب نمبر21	وراثت (Inheritance)
باب نمبر22	آپریٹرز اوورلوڈنگ
باب نمبر23	Inline، Virtual اور Friend فنکشنز
باب نمبر24	Templates، Exception Handling اور Polymorphism
باب نمبر25	اسٹریمز(Streams)

باب نمبر 1

کمپیوٹر پروگرامنگ کا تعارف

کمپیوٹر پروگرامنگ ایک ایسا لفظ جس سے آج کل تقریباً ہر شخص آشنا ہے۔ ++C لینگویج، کمپائیلر اور اس جیسی بہت سی اصطلاحات کو زیرِ بحث لانے سے پہلے پروگرامنگ پر کچھ بات ہو جائے: آج ہر طرف صرف و صرف ایک ہی نام ہے "کمپیوٹر پروگرامنگ" اور بس! یہ پروگرامنگ ہمارے لیے اتنی ضروری کب سے ہو گئی؟ تمام تر ملازمتوں کا دارومدار کمپیوٹر خواندگی (کمپیوٹر لٹریسی) پر بلاواسطہ یا بالواسطہ کیوں ہو گیا۔

ہمارے بزرگوں نے پروگرامنگ نہیں کی، 1400 سال پہلے پروگرامنگ کہاں تھی؟ حضرت آدمؑ کے زمانے میں پروگرامنگ کہاں تھی؟ تو کیا آج سے پہلے دنیا نہیں چل رہی تھی، چل رہی تھی تو کس طرح؟ اس کا سیدھا سا جواب ہے کہ پروگرامنگ ہر زمانے میں ہر جگہ موجود تھی ہے اور رہے گی، صرف طریقہ کار تبدیل ہوا ہے۔ پہلے پروگرامنگ مینوئل (Manual) ہوتی تھی۔ اب کمپیوٹرائزڈ ہو گئی ہے۔ ایسا کیوں؟ اس کا جواب آگے آئے گا، پہلے اتنا سمجھ لیجئے کہ پروگرام سے کیا مراد ہے؟ پروگرام کہتے ہیں "ترتیب وار کچھ مراحل کے عمل کو جس کے بعد کوئی نتیجہ اخذ ہو۔" دنیا کا ہر کام ایک مکمل پروگرام ہے۔ آیئے دیکھتے ہیں کس طرح؟ ہر پروگرام میں کچھ مرحلے یا Phases ہوتے ہیں:

1۔Input: وہ معلومات جو پروگرام چلنے سے پہلے ہمارے پاس موجود ہوں۔
2۔Process: ملنے والی معلومات کے تحت کچھ ترتیب وار اعمال جو ان معلومات کو حسبِ ضرورت کام میں لائیں یا ان کی Manipulation کریں۔
3۔Output: کئے گئے عمل کا نتیجہ یا رزلٹ۔

"Process" کے Phase کو ہم مزید 2 حصوں میں تقسیم کر سکتے ہیں:
اول: فیصلہ (Decision) اور؛ دوم: دوہرانا (Looping)

ہم نے کہا کہ دنیا کا ہر کام ایک مکمل پروگرام ہے تو دیکھتے ہیں کہ روزمرہ معمولات میں یہ تینوں Phases کہاں آتے ہیں۔ ایک آسان سا کام ہے۔ ہم چاہتے ہیں کہ سامنے پڑا ہوا مارکر اٹھالیں۔ یہ جو ہماری سوچ ہے، ہماری خواہش ہے یا جو ہم چاہ رہے ہیں یہ Input ہے جو ہمارے دماغ کی طرف منتقل ہوا۔

ہمارے دماغ سے کچھ سگنل فائر ہوئے جنہیں ہمارے ہاتھ نے موصول کیا۔ نیورونز کی مدد سے دماغ نے ہاتھ کو ہدایات دیں، ہاتھ آگے بڑھا اور انگلیوں نے خم کھا کر مارکر کو اپنی گرفت میں لے لیا۔ یہ تمام عمل Process تھا۔

مارکر ہمارے ہاتھ میں موجود ہے۔ یہ Output ہے۔

چلئے ایک اور مثال سے سمجھتے ہیں۔ ہمیں ایک کتاب خریدنی ہے جس کے بارے میں ہمیں مندرجہ ذیل معلومات ہیں:

نام:	C++ Programing
جلد:	2nd
مصنف:	Satish Jain
پبلشر:	BPB
دکان کا نام:	Venguard
پتا:	مشرق سینٹر

اب مندرجہ بالا تمام معلومات جو ہمارے دماغ میں موجود ہیں بطور Input استعمال ہوں گی۔ خیر! ان معلومات کو ذہن میں رکھتے ہوئے ہم کسی بھی ذریعے سے مشرق سینٹر میں Venguard بک اسٹور پر پہنچ گئے۔ یہ تمام عمل "Process" کا ہے۔ اب ہم نے بک شیلف میں سے ایک کتاب اُٹھائی اور اُس پر لکھی ہوئی معلومات کو اپنے دماغ میں موجود Input سے کمپیئر (موازنہ) کیا۔ اگر یہ دونوں یکساں (Same) ہیں تو ٹھیک؛ ورنہ ہم نے اُس کتاب کو رکھ کر دوسری کتاب اُٹھائی — اور یہی عمل بار بار دہراتے رہے۔ یہ ہے Looping۔ لوپنگ کہتے ہیں ایک ہی کام کو بار بار دہرانے کو۔
خیر! ہمیں اپنی من پسند کتاب مل گئی۔ اب ہم کتاب میں لکھی ہوئی قیمت کا موازنہ اپنی جیب میں موجود رقم سے کرتے ہیں۔ اگر کتاب کی قیمت زیادہ ہوئی تو ہم خرید نہیں سکتے، کم ہوئی تو ہم خرید سکتے ہیں، برابر ہوئی تو ہم واپسی کے کرائے کے بارے میں سوچیں گے۔ مزید براں ہم کوشش کریں گے کہ کچھ رعایت بھی مل جائے۔

یہ تمام عمل فیصلہ کرنے کا، یعنی Decision کا ہے۔ Desicion سے مراد یہ ہوئی کہ ہم مختلف صورتحال میں کیا فیصلہ کرتے ہیں۔
بالآخر، یا تو ہم کتاب خریدیں گے یا بغیر خریدے ہی واپس آجائیں گے۔ یہ Output ہوگا۔

اسی انداز میں سوچتے ہوئے آپ دنیا کے ہر کام کو دیکھ لیجئے، خواہ وہ کتنا ہی چھوٹا ہو یا بڑا۔ اُس میں یہ تمام مراحل بدرجہ اُتم موجود ہوں گے۔

بہرحال! اس تمام بحث کے بعد ہم اس نتیجے پر پہنچے ہیں کہ "پروگرام" کچھ مراحل کے ترتیب وار عمل کا نام ہے اور ان اعمال کو ہم تین حصوں میں درجہ بند (Classify) کرتے ہیں جنہیں ہم بالترتیب Input، Process اور Output کا نام دے رہے ہیں۔ یہ تو ہو گیا پروگرام۔ اب مزید کچھ سمجھنے سے پہلے کمپیوٹر پر کچھ بات کر لیں۔

پروگرامنگ کے نقطۂ نگاہ سے کمپیوٹر کیا ہے؟

اگر ہم بات کرتے ہیں پروگرامنگ کے نقطۂ نگاہ سے، تو کمپیوٹر کچھ نہیں ہے سوائے ایک مشینی ڈھانچے کے؛ جس میں ہم کچھ Input داخل کرتے ہیں جو بائنری نمبرز کی شکل میں ہوتا ہے۔ وہ Input مختلف تاروں سے گزرتا ہوا ہمارے سامنے ظاہر ہو جاتا ہے۔ کمپیوٹر ایک معصوم بچہ ہے جسے کچھ نہیں پتا۔ ہر چیز ہم اُس کو سمجھاتے ہیں اور جو سمجھاتے ہیں، جیسے سمجھاتے ہیں وہ اُس پر اُسی طرح عمل کرتا ہے۔ کمپیوٹر احساسات سے عاری ایک مشین کا نام ہے جس کے پاس کوئی جذبہ، کوئی احساس نہیں۔ اگر ہم چاہتے ہیں کہ اسکرین Clear ہو جائے تو جب تک ہم CLS لکھ کر Enter نہیں کریں گے، Clear نہیں ہو گی۔ اس کے برعکس ہم صبح سے شام تک کمپیوٹر کی منتیں کرتے رہیں مگر وہ اسکرین Clear نہیں کرے گا۔

مثلاً ہم نے ایک پروگرام بنایا ہے کہ جیسے ہی ہم Enter پریس کریں، کمپیوٹر فائر سکیورٹی گارڈز کو الارم بجا کر مطلع کر دے۔ اب ہم Enter پریس نہ کریں، کمپیوٹر کے سامنے پورا گھر جل جائے۔ مگران صاحب کے کان پر جُوں تک نہ رینگے گی کیونکہ یہ ہر قسم کے احساسات سے عاری ہیں... سوائے چند کمانڈز کے کچھ نہیں سمجھتے۔

نتیجہ: یہ نکلا کہ کمپیوٹر ایک مشین ہے جسے کچھ نہیں پتا۔ ہر چیز ہم بتاتے ہیں، اُس کو ہر بات سمجھاتے ہیں اور وہ اُس کے مطابق کام کرتا ہے۔

ابھی تک ہم نے یہ جانا کہ پروگرام کیا ہے اور کمپیوٹر کسے کہتے ہیں۔ اب آتے ہیں اس طرف کہ کمپیوٹر ہی کیوں؟ آخر ہم کمپیوٹر کیوں استعمال کرتے ہیں۔ جب تمام تر کام Manually ہو رہا تھا تو ہم نے اپنے کام کو کمپیوٹرائزڈ کیوں کیا؟ وہ کون سے ایسے فوائد ہیں جن کی بنا پر ہم نے اپنی Approach تبدیل کی۔

دیکھئے، مثلاً مجھے اکاؤنٹنگ آتی ہے۔ اب میں کیا کرتا ہوں کہ بہت سے پیسے خرچ کر کے ایک مشین (کمپیوٹر) لے آتا ہوں۔ اُسے تو کچھ نہیں آتا۔ اب کیا کروں، اب اُس کے لیے میں ایک سافٹ ویئر مزید رقم خرچ کر کے بنواتا ہوں یا دوسرے لفظوں میں اُسے اکاؤنٹنگ سکھاتا ہوں۔ پھر ایک آپریٹر کو ماہانہ تنخواہ پر نوکری دیتا ہوں کہ وہ اُسے چلا سکے۔ آخر اتنی دردسری کیوں؟ جب ایک کام سیدھے سادھے طریقے سے ہو رہا تھا تو کیوں ایک "بے وقوف مشین" خرید کر لایا اور اتنا خرچہ مزید کر رہا ہوں۔ بجلی، مرمت اور دیکھ بھال کا خرچہ الگ۔ وہ ایسے کیا فوائد ہیں جن کی وجہ سے ہم کمپیوٹر کو استعمال کر رہے ہیں؟ آیئے دیکھتے ہیں۔ مندرجہ ذیل تین وہ فوائد ہیں جن کی وجہ سے ہم کمپیوٹر کو انسان پر ترجیح دیتے ہیں:

1۔ رفتار (Speed)؛
2۔ گنجائش/جگہ (Capacity)؛ اور
3۔ صحیح/غلطی سے پاک (Accuracy)

1۔ رفتار: پہلا اور سب سے اہم فائدہ جو ہمیں کمپیوٹر کے استعمال سے ہو رہا ہے وہ ہے رفتار (Speed)۔ کمپیوٹر میں جو بائنری سگنلز گزر رہے ہیں، اُن کی رفتار لاکھوں کلومیٹر فی سیکنڈ کے حساب سے ہے؛ بالکل اسی طرح انسانوں میں بھی سگنل گزرنے کا عمل ہوتا ہے۔ جی ہاں! انسانی دماغ سے بقیہ جسم کی طرف سگنل پاس ہوتے ہیں جن کی رفتار دس ہزار میل فی سیکنڈ ہے۔ بس یہی وہ بنیادی نکتہ ہے جس کی وجہ سے کمپیوٹر کو انسانوں پر برتری حاصل ہے۔ اب کمپیوٹر کوئی فائل Save کر رہا ہے تو اپنی اُسی بے پناہ رفتار سے۔ اور اگر انسان کوئی چیز یاد کر رہا ہے تو وہی دس ہزار میل فی سیکنڈ کی رفتار سے۔ یہی معاملہ دوبارہ یاد کرنے اور Restore کے عمل میں ہو گا اور بالکل یہی بقیہ تمام کاموں میں۔ یاد رکھئے! رفتار وہ واحد پہلو ہے جس کی وجہ سے کمپیوٹر کو انسانوں پر برتری حاصل ہے۔

2۔ گنجائش: یعنی capacity کہتے ہیں جگہ کو۔ مطلب یہ کہ ہم کتنا ڈیٹا اسٹور کر سکتے ہیں۔ آیئے کمپیوٹر اور انسان کا موازنہ کرتے ہیں۔ عام طور پر (2003ء میں) جو ہارڈ ڈسکس ہمارے پاس دستیاب ہیں وہ ہیں 80 گیگا بائٹ یا اس سے بھی کچھ زیادہ کی ہوتی ہیں۔ سائنس دانوں نے تحقیق کی کہ دنیا کے کس آدمی نے اپنے دماغ کا سب سے زیادہ حصہ استعمال کیا ہے تو پتا چلا کہ "آئن اسٹائن" نے۔ اُس نے اپنے دماغ کا 9.7 فیصد حصہ استعمال کیا تھا۔ یعنی تقریباً دسواں حصہ استعمال کر کے انسان چاند، مریخ کو تسخیر کر چکا ہے۔ اگر مکمل کر لیا تو یقیناً وہ زمان و مکان سے آزاد ہو جائے گا۔ خیر! اس 9.7 فیصد کو Bytes میں ناپا گیا تو اس کی Capacity بنی 40 گیگا بائٹ۔ آپ کو تو معلوم ہی ہے کہ کمپیوٹر میں ایک Digit کا اضافہ کرنے سے Capacity ڈبل ہو جاتی ہے۔ مثلاً اگر 3 bits کی جگہ ہے تو ہم 7 اسٹور کر سکتے ہیں اور اگر 4 بٹس ہیں تو 15 اسٹور کر سکتے ہیں۔ بالکل یہی فارمولا انسانی دماغ پر بھی لاگو ہوتا ہے، انسانی دماغ کی مکمل Capacity آج تک معلوم نہیں کی جا سکی۔ مگر بات آ جاتی ہے پھر وہیں رفتار پر۔ یوں سمجھئے کہ کمپیوٹر کے پاس جگہ تو کم ہے مگر ہمارا data ہی اتنا زیادہ نہیں کہ ہمیں زیادہ جگہ کی ضرورت محسوس ہو۔ مزید یہ کہ کمپیوٹر اُس جگہ کو بے انتہا رفتار سے استعمال کرنے کے قابل ہے۔ اب انسان کے پاس جگہ تو بہت ہے مگر وہ بے چارہ اس رفتار سے اُسے استعمال ہی نہیں کر سکتا۔ اسی لئے کمپیوٹر کو انسانوں پر برتری حاصل ہے۔

Accuracy_3: کمپیوٹر ایک Accurate مشین ہے۔ بالکل ٹھیک کہا آپ نے۔ تو کیا حضرت انسان Accurate نہیں ہو سکتے۔ اگر نہیں تو کمپیوٹر کس طرح ہے۔ سیدھا سا جواب ہے کہ پہلا مسئلہ تو اسپیڈ ہی کا ہے۔ اگر آپ کو دو جمع دو کرنے ہوں اور آپ کے پاس 3 ماہ کا وقت ہو تو کیا آپ غلطی کریں گے! بالکل نہیں۔ انسان پھر مار کھاتا ہے اسپیڈ کے چکر میں۔ دوسری بات یہ کہ کمپیوٹر Accuate نہیں بلکہ وہ غلطی کر ہی نہیں سکتا... بالکل اسی طرح جیسے فرشتے گناہ نہیں کر سکتے۔ بھئی! کچھ تار ہیں، پہلے سرے سے data اِن کرو تو دوسرے سرے سے آؤٹ ہو جائے گا۔ اس کے علاوہ data کہیں جا ہی نہیں سکتا۔ تیسری بات یہ کہ کمپیوٹر accurate اس لئے ہے کہ وہ احساسات سے عاری ہے۔ انسان تھک بھی جاتا ہے، بیرونی ماحول کا اثر بھی اُس پر پڑتا ہے، اُسے بھوک بھی لگتی ہے، نیند بھی آتی ہے، جبکہ کمپیوٹر صاحب ان تمام چیزوں سے بے نیاز ہیں۔ مثلاً آپ کچھ حساب کتاب کر رہے ہوں اور باہر گلی میں فائرنگ ہو رہی ہو تو آپ کا تمام کام چوپٹ ہو جائے گا۔ برعکس اس کے اگر کمپیوٹر ایک ٹیبل کو Sort کر رہا ہو اور پورے گھر میں آگ لگ جائے تو اُس کی صحت پر کوئی اثر نہیں پڑے گا۔ وہ نہایت اطمینان سے File کو Sort کرتا رہے گا۔ ہم جب کمپیوٹر کو جذبات یا احساسات دیں گے وہ Accurate نہیں رہے گا۔

آپ مصنوعی ذہانت (آرٹی فیشل انٹیلی جنس) پر بنی ہوئی بہت سی فلمیں دیکھی ہوں گی جن میں روبوٹس انسانوں سے لڑ رہے ہوتے ہیں۔ یاد رکھئے! جب کمپیوٹر خود سوچنے سمجھنے کے قابل ہو جائے گا تو حکم کا غلام نہیں رہے گا۔ وہ پہلے اپنا فائدہ سوچے گا اور پھر کچھ اور۔ اس تمام بحث کا نتیجہ یہ نکلا کہ کمپیوٹر، Accurate اسی لیے ہے کہ وہ غلطی نہیں کر سکتا۔

انسان کو اعلیٰ ترین کمپیوٹر بھی کہا جاتا ہے۔ آسٹریلین سائنسدانوں نے دعویٰ کیا تھا کہ وہ ایسا کمپیوٹر بنا سکتے ہیں جو انسانی دماغ کے مساوی ہو اور وہ اتنی جدید ٹیکنالوجی استعمال کریں گے کہ صرف اعشاریہ ایک انچ کے رقبے پر 10 لاکھ میگا بائٹس اسٹور کریں گے۔ مگر اُس کمپیوٹر کا سائز (حجم) براعظم آسٹریلیا کے برابر ہوگا! دنیا میں آج تک کوئی لاجک گیٹ ایسا نہیں بن سکا جس میں ہم بیک وقت 4 سے زائد Input دے سکیں؛ جبکہ انسانی دماغ میں موجود ہر ہر خلئے میں 10 ہزار سے زائد ان پُٹ کنکشنز موجود ہیں۔

اس تمام بحث کا خلاصہ یہ نکلا کہ انسان ہر طرح کمپیوٹر سے زیادہ ہے مگر صرف رفتار ہی وہ واحد پہلو ہے جس کی وجہ سے ہم کمپیوٹر کو روزمرہ زندگی میں استعمال کر رہے ہیں۔

پروگرام کسے کہتے ہیں؟ کمپیوٹر کیا ہے؟ کمپیوٹر کو پروگرامنگ میں کیوں استعمال کر رہے ہیں؟ یہ تمام باتیں ہم نے سمجھیں۔ اب دیکھتے ہیں کہ کمپیوٹر پروگرامنگ سے کیا مراد ہے۔

آپ یقیناً اُکتا گئے ہوں گے کہ C++ کا کہہ کر آپ کے ساتھ کیا کیا جا رہا ہے۔ دل میں یقیناً راقم الحروف کے لئے اردو زبان کا بھرپور استعمال ہو رہا ہوگا۔ تو از راہ کرم تھوڑا صبر سے کام لیجئے۔ جب تک آپ ان بنیادوں کو نہیں سمجھیں گے، پروگرامنگ کو نہیں سمجھیں گے، پروگرام نہیں بنا سکیں گے۔ تھوڑا سا انتظار اور...

ہاں تو جناب! بات چل رہی تھی کمپیوٹر پروگرامنگ کی۔ تو جناب! کمپیوٹر پروگرام سے مراد ہے:

"بامعنی ہدایات کا مجموعہ" (Set of Well defined Statements)

یہاں پر مجموعے سے مراد ہے ایک سے زائد ہدایات بامعنی سے مراد یہ کہ جو بھی Command، Statement یا ہدایات ہم دے رہے ہیں وہ کمپیوٹر کے لیے بامعنی ہوں۔ دیکھئے مجھے اپنا نام Display کروانا ہے C++ میں۔ اب میں اگر Editor میں لکھتا ہوں کہ

Please write my name on Screen, My name is ABC

اب یہ جملہ بامعنی تو ہے، کس کے لیے؟ میرے اور آپ کے لیے۔ مگر یہاں بامعنی کا لفظ ہمارے اور آپ کے لیے نہیں بلکہ کمپیوٹر کے لیے ہے۔ بامعنی ہدایات ہوں مگر وہ کمپیوٹر کے لئے بامعنی ہوں؛ خواہ ہم سمجھیں یا نہ سمجھیں۔ اگر نام پرنٹ کروانا ہو تو وہی سادہ ترین ہدایت لکھیں گے جو اس لینگویج میں نام لکھنے کے لئے مخصوص ہو۔

اگر ہم کوئی پروگرام لکھنے کے بعد اُسے Run کرتے ہیں اور Output ہماری مرضی کا نہ ہو تو ہم کبھی بھی یہ نہیں کہتے کہ کمپیوٹر خراب ہو گیا ہے۔ ہم دوبارہ اپنے پروگرام میں تبدیلی کرتے ہیں، کیونکہ output تو صرف وہی آئے گا جو ہم نے لکھا ہے، آیا صحیح یا غلط! اب کرتے ہیں بات ہدایات (Statements) پر۔

Statement سے کیا مراد ہے؟

Statement کمپیوٹر سے **Communicate** کرنے کا، اُسے اپنی بات سمجھانے کا، اُسے مخاطب کرنے کا واحد ذریعہ ہے۔

انسانوں کے پاس ایک دوسرے سے رابطہ کرنے یعنی کمیونی کیشن کے مختلف ذرائع ہیں۔ جیسے ہمارے پاس مختلف زبانیں مثلاً اردو، انگلش، سندھی، پشتو، گجراتی وغیرہ ہیں؛ اسی طرح کمپیوٹر میں بھی مختلف لینگویجز ہیں BASIC، C++، Pascal، Cobol، RPG، Fortran وغیرہ۔

اس کے علاوہ انسان اپنے جذبات کی ترجمانی کے لئے اشاروں کے ذریعے بھی بات کر سکتے ہیں۔ جبکہ کمپیوٹر کے پاس احساسات نام کی کوئی چیز ہی نہیں۔ اگر کمپیوٹر سے کوئی کام کروانا ہے تو صرف اور صرف ایک ہی طریقہ ہے، اور وہ ہے Statement۔ مثلاً اگر آپ کے سامنے ایک شخص بیٹھا ہوا ہے تو آپ اسے دیکھ کر اندازہ لگا سکتے ہیں کہ وہ غمگین ہے، خوش ہے یا پریشان۔ ایک بچہ آپ کے سامنے شور کر رہا ہے، آپ اُسے گھور کر دیکھیں تو وہ خاموش ہو جائے گا۔

سیدھے سادھے طریقے سے یوں سمجھئے کہ جو حواسِ خمسہ انسانوں کے پاس ہیں، کمپیوٹر ان سے بے بہرہ ہے۔ اب آپ کچھ کمپیوٹر سے کروانا چاہ رہے ہیں تو اُسے صرف اُس زبان میں ہی بات کر سکتے ہیں جو وہ سمجھ رہا ہو اور وہ واحد ذریعہ ہے Statements۔ ہاں! تو جناب اب کمپیوٹر پروگرامنگ کی بھی تشریح ہو گئی۔

کمپیوٹر پروگرامنگ کی ابتداء

اب کمپیوٹر سے کام کروانا ہے تو اُس سے اُسی کی زبان میں بات کرنی پڑے گی اور ان حضرت کو سوائے 0,1 کے کچھ سمجھ میں ہی نہیں آتا۔ خیر 0,1 میں پروگرامنگ شروع کی، اپنی ہر بات کے لیے 0,1 کا مخصوص کوڈ بنایا گیا۔ Assembly جیسی پروگرامنگ لینگویجز مارکیٹ میں آئیں، مگر 0,1 میں پروگرامنگ جان کا عذاب ثابت ہوئی۔ ایک چھوٹے سے پروگرام کو لکھنے کے لیے ایک طویل پروگرامنگ کرنی پڑتی۔ اور اُس پہ طرّہ یہ کہ دو دن بعد پروگرام دیکھو تو کچھ سمجھ میں ہی نہ آئے۔ کوئی اور شخص دیکھے تو بے کسی کی زندہ تصویر بن جائے۔ کوئی غلطی تلاش کرنا ہو تو بڑی مشکل پیش آئے۔

پروگرامنگ کا ارتقاء

اب اس تمام طریقۂ کار کو چھوڑ کر حضرت انسان نے سوچا کہ ہم اپنی زبان میں ہی بات کریں۔ چھوٹے چھوٹے جملے یا الفاظ لکھیں اور کام ہو جائے۔ مگر اب ایک عدد جن درکار تھا جو ہماری باتیں کمپیوٹر کو سمجھا سکے...اور بالآخر وہ جن مل گیا۔ اُس کا نام کمپائیلر (Compiler) ہے۔ انسان کے لئے قابلِ فہم زبان میں دی گئی ہدایات کو Binary یا مشین لینگویج میں تبدیل کرنے کا تمام تر کام Compiler سرانجام دینے لگا۔

compiler کیا ہے؟

پاکستانی وزیرِاعظم چین گئے۔ اب چین کے وزیرِاعظم کو اُردو یا انگلش سے کبھی واسطہ نہیں پڑا تھا۔ پاکستانی وزیرِاعظم کو چین کی زبان نہیں آتی تھی۔ اب کیا ہوا کہ ایک شخص درمیان میں بیٹھ گیا جو اردو کے جملوں کو چین کی زبان میں اور چینی زبان میں ادا کئے گئے جملوں کو اردو میں ترجمہ کر کے بتانے لگا۔ بس compiler بھی اُسی مترجم کا کام انجام دے رہا ہے۔ یہ ایک ایسا پروگرام ہے جس کے پاس ایک ڈکشنری موجود ہے High level To low level جس کی مدد سے یہ ہماری لکھی ہوئی ہدایت کو مشین لینگویج میں تبدیل کر دیتا ہے۔ مشین لینگویجز کو ہم low level اور وہ لینگویجز جن میں ہم کی ورڈز یا انگلش جملوں کی شکل میں پروگرامنگ کرتے ہیں، high level لینگویجز کہتے ہیں۔ جیسے Pascal وغیرہ۔

باب نمبر 2

پروگرامنگ لینگویجز کی تاریخ اور ++C کی ابتداء

1960ء میں انٹرنیشنل کمیٹی نے ایک کمپیوٹر لینگویج وضع کی جس کا نام Algol رکھا گیا۔ 1967ء میں مارکیٹ میں آنے والی لینگویج کا نام تھا BASIC Computer Programing Language جسے مارٹن رچرڈز نے AT&T Bell Lab میں بنایا۔ 1970ء میں کین تھامپسن نے AT&T Bell lab میں B لینگویج تخلیق کی اور 1972ء میں AT&T ہی کے ڈینس رچی نے مشہور زمانہ C لینگویج تخلیق کی۔

آپ کو یہ اتنے سارے نام اور تاریخیں بتانے کا مقصد ہرگز یہ نہیں تھا کہ آپ کو تاریخ سے کچھ رغبت دلائی جائے بلکہ اس کا اصل مقصد اُس "C" کو واضح کرنا تھا جو اس لینگویج کے نام میں آتا ہے۔ Algol کو A لینگویج کہا جاتا تھا، بعد میں B آئی، اب جب ایک نئی لینگویج مزید خصوصیات کے ساتھ وضع کی گئی تو اسی ترتیب کو جاری رکھتے ہوئے، نئی تخلیق ہونے والی لینگویج کا نام C رکھا دیا گیا۔ یاد رکھئے، C کسی بھی لفظ کا مخفف، شارٹ فارم یا Abbreviation نہیں۔

1983ء میں AT&T Bell labs کے ہی پروگرامر اسٹراؤس ٹرپ نے "C" میں بہت سی نئی خصوصیات شامل کر دیں اور اس کا نام رکھ دیا ++C۔

اب "C" تو سمجھ میں آتا ہے۔ یہ دو عدد پلس (++) کی منطق کیا ہے؟ پلس پلس ہی کیوں؟ اس کی 3 وجوہ ہیں۔ یہ دو پلس 3 چیزوں کو ظاہر کرتے ہیں:

اوّل: اس لینگویج کے دونوں تخلیق کاروں کا تعلق اہلِ یونان سے ہے۔ جس طرح ہمیں اردو میں کسی جملے پر زور دینا ہوتا ہے یا اُس کی اہمیت کو اُجاگر کرنا ہوتا ہے تو ہم وہ جملہ بار بار دہراتے ہیں یا لکھتے ہیں (عموماً تقریروں میں)۔ جس طرح کسی ٹیکسٹ پر کسی اہم ہیڈنگ کو ہم بولڈ، اٹالک یا underline کر دیتے ہیں، بالکل اسی طرح یونانی زبان میں کسی جملے کو اُجاگر کرنے کے لئے اُس کے آگے ایک مرتبہ + لکھ دیا جاتا ہے۔ اب موصوف (تخلیق کار) اپنی تخلیق پر کچھ زیادہ ہی جذباتی ہو گئے اور اُنہوں نے ایک چھوڑ دو دو پلس لکھ دیئے۔ ماحصل یہ نکلا کہ بہت زیادہ اہمیت والی لینگویج۔

دوم: پروگرامنگ کرتے ہوئے ہم مختلف طریقہ کار (اپروچز) کو مدِ نظر رکھتے ہیں، گہرائی میں غوطہ لگائے بغیر صرف فی الحال اتنا سمجھ لیجئے کہ C بیک وقت دو اپروچز کو Support کرتی ہے: اسٹرکچرڈ (Structured) اپروچ کو بھی اور آبجیکٹ اورینٹڈ (Object Oriented) کو بھی۔ اب دو پلس، دو اپروچز کو سپورٹ کرنے والی خصوصیت کو ظاہر کرتے ہیں۔

سوم: تخلیق کار نے ایک پندرہ سالہ لینگویج Simula 67 جو کہ 1968ء میں منظرِ عام پر آئی (پہلی آبجیکٹ اورینٹڈ لینگویج) اور C، دونوں کو ملا کر ++C بنائی۔ اب دو پلس دو مختلف لینگویجز کو ظاہر کر رہے ہیں۔

ان تین باتوں کے علاوہ ایک چوتھی بات جو اور کہی جاتی ہے، وہ اس لینگویج کے Syntax کے بارے میں ہے۔ جب ہم C لینگویج میں کسی متغیر (Variable) کی قیمت میں 1 کا اضافہ کرنا چاہتے ہیں تو اس کے آگے ++ لکھتے ہیں مثلاً ++A کا مطلب ہوگا:

A=A+1

چنانچہ C میں اضافے سے بننے والی لینگویج کو ++C کہا گیا۔ چلیئے ++C تو واضح ہوا۔

باب نمبر 3

بنیادی اصطلاحات اور ++C کا پہلا پروگرام

پروگرامنگ Fundamentals

اب ہم ++C پروگرامنگ کی ابتداء کرتے ہیں۔ سب سے پہلے دو باتوں کو ذہن میں بٹھا لیجئے، آپ کی تمام تر پروگرامنگ کا دارومدار انہی پر ہوگا۔

اوّل: کمانڈز لکھنے کا طریقہ کار یا Syntax

دوم: ترتیب یا sequence

Syntax -1: کسی بھی لینگویج کا ایک طریقہ کار یا Syntax ہوتا ہے۔ ایک اجتماعی Syntax اور ایک ہر کمانڈ کا انفرادی Syntax۔ آپ کوئی بھی لینگویج سیکھنے کی کوشش کریں۔ سب سے پہلے آپ کو اُس کی Statements کا Syntax سیکھنا ہوگا کہ ایک Statement کس طرح لکھی جائے۔ اُس کی Spelling، مختلف Parameters کی ترتیب وغیرہ۔

Sequence -2: ترتیب۔ جی ہاں! ترتیب کسی بھی پروگرام کی مکمل کامیابی میں سب سے اہم کردار ادا کرتی ہے۔ اگر آپ نے پروگرام لکھتے وقت ترتیب کا خیال نہ رکھا تو یاد رکھیں آپ کے Output کا بیڑہ غرق ہو جائے گا۔ ہوسکتا ہے کہ Compiler آپ کو کسی غلطی کی نشاندہی نہ کرائے مگر output من پسند نہیں آئے گا۔

یاد رکھیں! کمپیوٹر پروگرام کا output کبھی بھی وہ نہیں آتا جو آپ چاہ رہے ہیں یا جو آپ نے لکھا ہوا ہوگا، output ہمیشہ وہ آئے گا جو computer نے سمجھا۔ اسی لیے ہمیں سب سے پہلے Compiler کے مزاج کو سمجھنا پڑے گا۔ بس مندرجہ بالا دو باتوں کا خیال رکھئے۔ Compiler آپ کا یا آپ Compiler کے مزاج آشنا ہو جائیں گے۔

آپ میں سے بہت سے قارئین Syntax کی افادیت کو تو سمجھ چکے ہوں گے مگر sequence یقیناً واضح نہیں ہو رہا ہوگا۔ آیئے مثالوں سے سمجھتے ہیں۔ یاد رکھئے کہ جتنی اہمیت کسی بھی لینگویج میں Syntax کی ہے اُس سے کہیں زیادہ sequence کی ہے۔

مثال نمبر 1: ایک سادہ سا جملہ دیکھئے: ''پاکستان 1947ء میں بنا''۔
اس جملے میں حروف تہجی کو کس طرح ملانا ہے، یہ تو Syntax ہے۔
اب ذرا sequence کو تبدیل کر دیتے ہیں۔
''1947ء پاکستان میں بنا''
اب Syntax تو بالکل ٹھیک ہے مگر مطلب کا بیڑہ غرق ہو گیا۔

مثال نمبر 2: ہم نماز پڑھتے ہیں۔ رکوع، سجدے، ان میں پڑھی جانے والی تسبیحات، اُن کی تعداد، قیام، تشہد قرأت کرتے ہیں۔ میں صرف sequence کو معمولی سا تبدیل کر دیتا ہوں، سجدے پہلے کر لیتا ہوں اور رکوع بعد میں۔
ہاں! تو جناب، کیا خیال ہے نماز قابلِ قبول ہوگی؟ نہیں۔ مگر کیوں؟ Syntax تو بالکل ٹھیک تھے۔

ایک پروگرام بنایئے۔ پہلے نام پرنٹ کرائیں پھر اسکرین کلیئر۔ جب بھی پروگرام چلائیں گے اسکرین صاف نظر آئے گی۔ نام کہاں گیا؟ آپ نے sequence کا خیال نہیں رکھا۔ ذہن نشین کر لیجئے کہ sequence کی اہمیت کسی بھی طور پر کم نہیں۔

ہاں تو ++C کی طرف آجایئے۔ اب تک یقیناً آپ کے صبر کا پیمانہ لبریز ہو چکا ہوگا اور آپ کی نیکیوں کی ایک کثیر مقدار میرے نامہ اعمال میں منتقل ہو چکی ہوگی۔ لہٰذا بغیر کسی تشریح کے ++C کا پہلا پروگرام نوٹ کر لیجئے، پھر اُس کو سمجھ لیتے ہیں:

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
clrscr();
printf("My First Program");
getch();
}
```

یہ پروگرام آپ لکھیں گے IDE   C++ پر۔ IDE کا مطلب ہے Integrated Development Environment۔ کسی بھی PATH پر جہاں آپ نے ہارڈ ڈسک پر C++ انسٹال کی ہے، وہاں جا کر TC لکھ کر Enter پریس کر دیجئے اور IDE آپ کے سامنے آ جائے گا۔

بقیہ IDE کے Menu وہی ہیں، جن کا واسطہ آپ سے ہزاروں بار پڑ چکا ہوگا۔ اب آپ File کے Menu پر جائیے اور New پر کلک کر دیجئے۔ ایک نئی ونڈو آپ کے سامنے آ جائے گی جہاں آپ مندرجہ بالا پروگرام ٹائپ کر دیجئے۔ C++ میں جب تک آپ خود سے غلطی نہ کرنا چاہیں، کمپائلر آپ کو بالکل تنگ نہیں کرے گا۔ C++ پروگرامز میں ہونے والی غلطیوں میں سے 80% اسپیلنگ کی غلطیاں ہوتی ہیں۔

آیئے پروگرام کی وضاحت کر لیتے ہیں۔ سب سے پہلے سمجھتے ہیں پہلی Statement کو۔

include <stdio.h>

اس طرح کی Statements کے چار حصے ہوتے ہیں:

 # Buffer / channel sign

لفظ **Include**

<>less than or greater than sign

stdio.h header file

(.h) ایکس ٹینشن سے ہیڈر فائل کی

ہیڈر فائل: سب سے پہلے header file کو سمجھتے ہیں۔ ہیڈر فائل مجموعہ ہے ملتے جلتے (کام کے اعتبار سے) بہت سارے فنکشنز کا۔

C کے اندر ہم یہ والی Statement نہیں لکھتے کیونکہ C میں کُل Functions کی تعداد بہت کم تھی۔ جب بھی ہم C لوڈ کرتے تھے تو تمام Functions میموری میں لوڈ ہو جاتے تھے۔ مگر C++ میں Library Functions کی تعداد 3 ہزار سے بھی تجاوز کر گئی۔ اب اگر ہمیں صرف نام پرنٹ کروانا ہو تو کیا ضرورت ہے کہ ہم گرافکس کے یا ریاضی کے Functions کو Load کریں۔ لہٰذا C++ کے لائبریری فنکشنز کو مختلف فائلوں میں درجہ بند، یا Categorize کر دیا گیا اور ہر فائل کے اندر بہت سارے فنی اعتبار سے ملتے جلتے Functions رکھ دیے گئے۔ اب ہمیں جس قسم کے Functions درکار ہوتے ہیں اُنہی کو ہم میموری میں لوڈ کر لیتے ہیں۔

Function کیا ہوتے ہیں؟ ابھی صرف اتنا سمجھ لیجئے: ''ہدایات کا وہ مجموعہ جو کوئی منتخب نتیجہ دے، اُسے ہم فنکشن کہتے ہیں۔'' یہ چھوٹے چھوٹے بہت سارے پروگرامز ہوتے ہیں جنہیں ہم بڑے بڑے پروگرامز بناتے ہوئے استعمال کرتے ہیں۔ اب سوال یہ پیدا ہوتا ہے کہ آخر اتنی دردسری کیوں؟ ارے بھئی میموری کو بچانے کے لئے! تاکہ میموری کا کم از کم حصہ خرچ ہو۔ ایک مثال سے سمجھتے ہیں:

گھر کے اندر مختلف باکس یا ڈبے ہوتے ہیں۔ جیولری باکس، میک اپ باکس، فرسٹ ایڈ باکس، ڈرائنگ باکس وغیرہ۔ اب اگر ہمیں ڈرائنگ کرنی ہے تو ہم ڈرائنگ باکس اُٹھا کر اپنے سامنے رکھ لیتے ہیں تاکہ ضرورت کی کسی بھی چیز کو اُٹھا کر استعمال کر سکیں اور بہ آسانی واپس رکھ سکیں۔ اب اس باکس میں تمام تر اشیاء کا تعلق فنِ مصوری سے ہوگا۔ انجکشن یا لپ اسٹک اس باکس میں نہیں ہوگی۔ ایسا کیوں کیا گیا؟ آسانی کے لیے تاکہ جیسا کام درپیش ہو وہی باکس استعمال کیا جا سکے۔

بالکل یہی کام ہم نے C++ میں کیا ہے۔ ہر Header file ایک مکمل باکس ہے جس میں Related Functions ہیں۔ چلیں Header file کا مسئلہ واضح ہوا۔

stdio مخفف ہے Standard Input/output کا یعنی ''اسٹینڈرڈ اِن پٹ آؤٹ پٹ'' سے متعلق تمام فنکشن اس میں ہیں۔

conio مخفف ہے Console input/output کا۔

Console کمپیوٹر کی زبان میں اسکرین کو کہتے ہیں۔ جس طرح اسکرین کے لئے ہمیں اور کئی مختلف نام ملتے ہیں۔ مثلاً مانیٹر، FRT ڈسپلے اسکرین، Console، وڈیو ڈسپلے یونٹ، وڈیو ڈسپلے ٹرمینل وغیرہ۔

<>: یہ ڈیفالٹ Syntax ہے۔ ہیڈر فائل کے نام کے لیے آپ انورٹڈ کوما (" ") بھی استعمال کر سکتے ہیں۔ جیسے:

#include "stdio.h"

include یعنی شامل کرو۔ یعنی آگے بتائی ہوئی HeaderFile کو ہارڈ ڈسک پر سے اُٹھا کر Memory میں لوڈ کر لو۔

Buffer فرق کرتا ہے ڈائریکشن اور اسٹیٹمنٹ میں۔ (ڈائریکشن کا مطلب ہوتا ہے ہدایت جبکہ Statement کا مطلب ہے کام۔)

کے ساتھ آپ ہدایت دیتے ہیں۔

ہاں! تو جناب اب اس مکمل Statement کا مطلب ہوا: ''دی ہوئی Header file کو میموری میں لوڈ کر لو۔''

void main(void) کیا ہے؟ اس کا تفصیلی ذکر ہم فنکشن کے باب میں کریں گے۔ فی الحال اتنا سمجھ لیجئے کہ یہ main فنکشن ہے جو C++ کمپائیلر سب سے پہلے پڑھتا ہے اور آپ کو ہر پروگرام میں یہ لائن لازمی لکھنی ہے۔

کرلی بریکٹس ({ }) ظاہر کرتے ہیں پروگرام کی ابتدا اور خاتمے کو۔

clrscr فنکشن، اسکرین کو کلیئر کرتا ہے۔

printf فنکشن کی مدد سے ہم کسی بھی text یا variable کی Value پرنٹ کروا سکتے ہیں۔

اور getch کا مطلب ہے get character ۔ یہ یوزر اسکرین کو (جہاں آؤٹ پٹ ظاہر ہوتا ہے) تب تک روکے رکھتا ہے جب تک کہ آپ کوئی بٹن پریس نہ کردیں۔ ورنہ آپ فوراً IDE میں واپس آ جاتے ہیں اور آپ کو Output نظر نہیں آتا۔

آپ Header file کو Include نہ کروائیے۔ مثلاً conio.h کو تو clrscr اور getch پر Error آئے گا کہ

"Function should have a prototype"

آسان الفاظ میں یوں سمجھ لیجئے کہ فنکشن پہلے سے معلوم ہی نہیں کہ وہ header file میں ہے یا User defined ہے۔

ہاں! تو جناب پہلا پروگرام مکمل ہوا۔ ایک بات یاد رکھئے کہ C++ ایک Case sensitive لینگویج ہے۔ تمام ترہدایات آپ کو lower case میں ہی لکھنا ہوں گی۔

باب نمبر 4

ویری ایبلز اور ڈیٹا ٹائپس

VARIABLE کیا ہے؟

آسان الفاظ میں یوں سمجھ لیں کہ VARIABLE، میموری MEMORY میں موجود اُس جگہ (Segment) یا باکس کا نام ہے جہاں پر ہم کوئی ڈیٹا محفوظ کرتے ہیں۔ اب میموری کے ایڈریس تو Binary فارمیٹ میں ہوتے ہیں۔ اُن کو یاد کرنا کچھ آسان نہیں اور VARIABLE اُس وقت بنتے ہیں جب پروگرام چلتا ہے لہٰذا پروگرام چلنے سے قبل ہمیں ایڈریس بھی معلوم نہیں ہوتا۔

اس لیے ہم میموری کی اُس مخصوص جگہ (VARIABLE) کو جہاں ڈیٹا اسٹور کرنا ہوتا ہے، ایک نام دیتے ہیں تاکہ پروگرام میں جہاں کہیں بھی اُسے استعمال کرنا ہو، اُس نام کی مدد سے کیا جا سکے۔ آئیے اب دیکھتے ہیں کہ C++ میں VARIABLE کو ہم کس طرح Create (تخلیق) کر سکتے ہیں۔

:SYNTAX

```
Datatype     Variablename;
```

ڈیٹا ٹائپ   ویری ایبل کا نام

سب سے پہلے ہم ڈیٹا ٹائپ لکھیں گے پھر Space دیں گے اور اس کے بعد ویری ایبل کا کوئی بھی من پسند نام دے دیں گے۔ کوشش کیا کریں کہ ویری ایبل کا نام بامعنی ہو، یعنی اگر آپ AGE ان پٹ لے رہے ہوں تو نام بھی AGE ہی رکھیں۔

ویری ایبل تخلیق کرنے کے قواعد و ضوابط

1۔ ویری ایبل کا نام کسی الفابیٹ سے شروع ہو رہا ہو۔

2۔ الفابیٹ، نمبرز اور صرف ایک اسپیشل Symbol انڈر سکور (_) کے علاوہ کوئی کیریکٹر، نام میں استعمال نہیں کر سکتے۔

3۔ کوما، Space یا کوئی بھی Special Symbol استعمال نہیں کر سکتے۔

4۔ سافٹ ویئر معیارات (SOFTWARE STANDARDS) کے تحت ویری ایبل کا نام آٹھ کیریکٹرز سے زیادہ کا نہ ہونا چاہیے مگر ٹربو C++ آپ کو 63 اور کچھ کمپائلرز 40 کیریکٹرز تک Allow کرتے ہیں۔

آئیے مثالوں سے سمجھیں۔ مندرجہ ذیل کچھ ویری ایبلز ہیں:

1- _ _ _ _ _
2- A_B
3- C D
4- 4season
5- A,B,
6- A$B
7- A=B
8- A~

مندرجہ بالا ناموں میں 1 اور 2 صحیح ہیں باقی تمام غلط ہیں۔ نمبر 3 میں ہم نے SPACE کو استعمال کیا ہے جو جائز نہیں۔ نمبر 4 میں پہلا کیریکٹر نیومیرک ہے، نمبر 5، 6، 7 اور 8 میں بالترتیب کوما، ڈالر $، برابر = اور ٹائلڈ کیریکٹر استعمال ہوئے ہیں۔ یہ تمام اسپیشل کیریکٹرز ہیں جن کا استعمال ناموں میں ممنوع ہے۔ یقیناً ویری ایبل اور ان کے ناموں کو رکھنے کا طریقہ کار واضح ہو گیا ہوگا۔ اب دیکھتے ہیں ڈیٹا ٹائپ سے کیا مراد ہے۔

ڈیٹا ٹائپ (DATA TYPE)

جیسا کہ نام سے ظاہر ہے، ڈیٹا ٹائپ سے مراد ویری ایبل میں محفوظ ہونے والی انفارمیشن کی قسم سے ہے، یعنی کس قسم کا ڈیٹا ہم میموری میں اسٹور کریں گے۔

کیا ڈیٹا کیریکٹر ہے، نیومیرک ہے، اسٹرنگ (String) ہے یا فلوٹ (Float)۔ ڈیٹا ٹائپ واضح کر دینے سے کمپائلر کو ان پر کیے جانے والے Operations میں مدد ملتی ہے۔ مثال کے طور پر دو عدد کیریکٹرز ہیں جن کی ڈیٹا ٹائپ بھی کیریکٹر ہے '1' اور '1'۔ اب اگر ان دونوں کو جمع کیا جائے تو جواب آئے گا

'1' + '1' = "11"

کمپائلر نے دونوں کو کیریکٹرز کے طور پر ٹریٹ کیا اور کونکیٹ (Concat) کر دیا۔ اب اگر دونوں کیریکٹرز کی ڈیٹا ٹائپ نیومیرک ہوتی تو جواب آتا۔

1 + 1 = 2

ڈیٹا ٹائپ کی مدد سے ہم یہ جان سکتے ہیں کہ ایک ویری ایبل میں موجود ڈیٹا یا VALUE پر ہم کون کون سے Operations کر سکتے ہیں اور کمپائلر ان کو کس طرح حل کرے گا۔ مزید یہ کہ ویری ایبل میموری میں کتنے BYTES پر اسٹور ہوگا اور زیادہ سے زیادہ کون سی VALUE ہم اس پر اسٹور کروا سکتے ہیں۔ یہ تو تھا ڈیٹا ٹائپ کا بنیادی مقصد۔

اب آیئے اس Concept کو کچھ تفصیل سے سمجھ لیتے ہیں۔ مندرجہ ذیل چارٹ کو دیکھئے۔

Types of Datatypes
1- Numeric
    1.1- Integer
    1.2- Real
2- Non Numeric
    2.1- Alphabetic
    2.2- Alphanumeric
3- Boolean

روزمرہ زندگی میں ہمارا واسطہ مذکورہ بالا ڈیٹا ٹائپس سے پڑتا ہے۔ یا تو انفارمیشن نیومیرک ہوگی یعنی ڈیٹا اعداد پر مشتمل ہوگا، یا وہ کیریکٹرز پر مشتمل ہوگا، یا ان دونوں کی ملی جلی شکل میں ہوگا۔ مثال کے طور پر عمر، تنخواہ، کمیشن، بجلی اور گیس وغیرہ کا بل، بینک بیلنس، تناسب یہ تمام نیومیرک ڈیٹا کی مثالیں ہیں۔ نام، پیغام، شہر، ملک، ذات یہ تمام نان نیومیرک کی مثالیں ہیں اور پتا، شناختی کارڈ نمبر یہ دونوں کی ملی جلی شکل ہے۔

نام:	ذیشان الحسن (نان نیومیرک)
ذات:	عثمانی (نان نیومیرک)
عمر:	21 (نیومیرک)
قد:	5'10" (نیومیرک)
شناختی کارڈ نمبر:	409-78-319881 (الفانیومیرک)

اب نیومیرک ڈیٹا کو ہم مزید تقسیم (Classify) کرتے ہیں انٹیجر (Integer) اور ریئل (Real) ڈیٹا میں۔

Integer سے مراد وہ اعداد ہیں جن میں Decimal point نہیں آتا مثلاً: 12, 15, 3, 1000 وغیرہ۔
اس کا استعمال ہم عمر، شناختی کارڈ نمبر وغیرہ میں کرتے ہیں۔

Real سے مراد وہ اعداد ہیں جن میں Decimal Point شامل ہو مثلاً 3.50, 4.5965, 3.146 وغیرہ۔
اس کا استعمال ہم تنخواہ، کمیشن، درجہ حرارت وغیرہ میں کرتے ہیں۔

اب رہا سوال **Boolean** کا۔ یہ ڈیٹا ٹائپ جارج بول نامی سائنس دان کے نام پر ہے۔ جارج بول نے سب سے پہلے Boolean Laws وضع کیے جو کہ کمپیوٹر کی بنیاد ہیں۔ اس قسم کے ڈیٹا میں ہمارے پاس صرف دو باتیں ہوتی ہیں مثلاً: صفر یا ایک، حاضر یا غائب، زندہ یا مردہ، پڑھا لکھا یا جاہل، دن یا رات، تندرست یا بیمار، غیر شادی شدہ یا شادی شدہ، پاس یا فیل، قریب یا دور، وغیرہ۔

اب دیکھتے ہیں ان ڈیٹا ٹائپ کو محفوظ رکھنے کے لیے C++ میں کون سی DATA TYPES ہیں۔ مندرجہ ذیل چارٹ ملاحظہ فرمایئے:

C++ Datatypes

1- Numeric
    1.1- Integer
    int, short, long
    1.2- Real
    float, double, long double

2- Non Numeric
    2.1- Alphabetic
    char
    2.2- Alphanumeric
    char

3- Boolean
    Not present

نیومیرک ڈیٹا میں Intger کے لیے short,int اور long نام کی ڈیٹا ٹائپس ہیں۔ Real کے لیے float, double, longdouble جبکہ الفابیٹک اور الفانیومیرک دونوں کے لیے char ڈیٹا ٹائپ ہے۔ Boolean کے لیے C++ میں کوئی بھی Built-In ڈیٹا ٹائپ نہیں ہے۔ Boolean کو ہم ڈائریکٹ کنٹرول کرتے ہیں لاجک کی مدد سے۔

Turbo C++ کا ورژن 3.0 جو کہ ہم پڑھ رہے ہیں، اس میں Boolean کے لیے کوئی ڈیٹا ٹائپ نہیں ہے۔ جبکہ ٹربو C++ ورژن 4.0 اور C# میں Boolean کے لیے Bool نامی ڈیٹا ٹائپ موجود ہے۔
Decision making یا مختلف صورت حال میں کوئی ایک کمانڈ یا کمانڈز کا مجموعہ منتخب کرنے کی صورت میں ہمیں اس قسم کی ڈیٹا ٹائپ کی ضرورت پڑتی ہے۔

یہاں تک تو بات سمجھ میں آ گئی ہوگی۔اب رہا سوال یہ کہ جب int ڈیٹا ٹائپ موجود تھی تو long کا مقصد کیا ہے؟ دونوں میں فرق ہے Bytes کا۔ int کا ویری ایبل میموری میں 2 Bytes لے گا جبکہ Long کا 4۔

جس قسم کا ڈیٹا درکار ہوتا ہے ہم اُسی کی مناسبت سے ڈیٹا ٹائپ کا انتخاب کرتے ہیں۔اب int کے اندر 32768 تک کی VALUE آ سکتی ہے۔فرض کیجئے کہ ہمیں عمر Age کا ان پُٹ لینا ہے۔اب ظاہر بات ہے کہ 32 ہزار سے زیادہ عمر کیا ہوگی تو ہم کیوں چار بائٹس کی جگہ استعمال کریں۔ہاں! اگر حساب کتاب لاکھوں کروڑوں میں ہے تو ہم long کو استعمال کریں گے۔

Datatypes کی رینج اور Bytes کو دیکھنے سے پہلے ویری ایبل تخلیق (Create) کرنے کی چند مثالیں سمجھ لیں۔

Syntax

```
Data Type   variable name
1- int      i ;
2- double   d;
3-float          f,g,k,i;
4-char           str1,str2;
5-long           l1;
```

ویری ایبل کے ناموں کے درمیان کوما (,) فرق کرتا ہے دو مختلف ناموں میں۔اس طریقے سے ہم ایک ہی لائن میں ایک ہی ڈیٹا ٹائپ کے ایک سے زائد ویری ایبل بنا سکتے ہیں۔

Escape Sequences

اسکیپ سیکوئینسز (Escape Sequences) مجموعہ ہیں دو کیریکٹرز کا: ایک بیک سلیش (\) اور دوم کوڈ۔کیریکٹر بیک سلیش (\) کمپائیلر کو Indicate کرتا ہے یا بتاتا ہے کہ اس کے آگے لکھا ہوا کیریکٹر، اسٹرنگ یا Message کا حصہ نہیں ہے بلکہ وہ ایک Sequence ہے۔

Escape Sequences سے مراد ایسے کیریکٹرز ہیں جو بذاتِ خود اسکرین پر پرنٹ نہیں ہوتے بلکہ ان کا کوئی نہ کوئی اثر اسکرین پر ہوتا ہے۔ان کو عموماً ہم لائن یا کالم وغیرہ کی ترتیب کے لیے استعمال کرتے ہیں۔عام طور پر استعمال ہونے والے Esape sequencers مندرجہ ذیل ہیں

```
\n    next line or new line
\r    return
\a    Alert
\t    Tab
\b    Backspace
```

اگر ہم Message یا Text کے ساتھ \n لکھ دیں گے تو کرسر، یوزر اسکرین پر (جہاں output ظاہر ہوتا ہے) اگلی لائن پر منتقل ہو جاتا ہے۔ \t استعمال ہوتا ہے Tab کے لیے؛ یعنی سات کیریکٹرز یا کالم کا space آ جائے گا۔ \a کا مطلب ہے الرٹ؛ کمپائیلر ایک Beep دے گا۔ \b ایک کیریکٹر پیچھے جانے کے لیے۔اور \r ہم استعمال کرتے ہیں Enter کے بٹن کے لیے۔

آیئے escape sequencers کو مثالوں سے سمجھتے ہیں۔

```
printf("ABC\nXYZ");
```

اس Statement کا output آئے گا:

```
ABC
XYZ
```

\n فوراً کرسر کو اگلی لائن پر منتقل کر دے گا۔

```
printf("ABC\n\tXYZ");
```

اس پروگرام کا آؤٹ پٹ کچھ ایسے نظر آئے گا:

```
ABC

        XYZ
```

اب ایک مثال اور ملاحظہ فرمایئے:

```
printf ("ABC\n\n\nxy\tz ");
```

اور اس کمانڈ کا آؤٹ پٹ ایسا ہوگا:

```
ABC



xy      z
```

باب نمبر 5

Operators اور Format Specifiers

فارمیٹ اسپیسی فائرز (Format Specifiers)

Format specifier دو کیریکٹرز پر مشتمل ہوتا ہے، ایک (%)Percentage کا سائن اور دوسرا کوڈ کیریکٹر۔
Percentage سمبل کمپائلر کو بتاتا ہے کہ یہ Sequence اسٹرنگ یا ٹیکسٹ کا حصہ نہیں ہے اور کوڈ کیریکٹر کمپائلر کو بتاتا ہے کہ ڈیٹا کو کس طرح Process کیا جائے، کس طرح ڈسپلے کیا جائے یا کس طرح میموری میں محفوظ کیا جائے۔
عموماً جو Format Specifiers ہم استعمال کرتے ہیں وہ مندرجہ ذیل ہیں۔

%d **Decimal /integer**
%f **Float**
%c **Character**
%s **String**
%e **Exponent**

آیئے، ان کا استعمال سمجھ لیتے ہیں۔ d% ہم استعمال کرتے ہیں ڈیٹا یا ویلیو کو Decimal فارمیٹ میں پرنٹ کروانے کے لیے۔ اگر آپ ()printf کے ساتھ کسی ویری ایبل کی ویلیو پرنٹ کروانا چاہتے ہیں تو آپ کو لازماً بتانا پڑے گا کہ اُس کا فارمیٹ کیا ہو۔ مندرجہ ذیل مثالیں دیکھئے:

```
printf("%d" , 'A');
```

اس کمانڈ کا output آئے گا 65۔ جی ہاں! ہم نے کیریکٹر 'A' کو Decimal فارمیٹ کے ساتھ پرنٹ کروایا ہے۔
لہٰذا 'A' کا متبادل ASCII کوڈ پرنٹ ہو جائے گا۔ بالکل اسی طرح

```
printf("%c" , 65);
```

کا output آئے گا A۔ کیونکہ ہم نے 65 کو کیریکٹر فارمیٹ میں پرنٹ کروایا ہے۔ عموماً ہم فارمیٹ Specifiers ڈیٹا ٹائپ کی مناسبت سے استعمال کرتے ہیں، جیسی ویری ایبل کی ڈیٹا ٹائپ ہوتی ہے ویسا ہی Specifier استعمال کیا جاتا ہے۔

scanf()

() scanf (اسکین ایف) ++C میں موجود ایک فنکشن کا نام ہے جو stdio.h میں ہوتا ہے۔
اس فنکشن کی مدد سے ہم یوزر سے input لیتے ہیں۔ سب سے پہلے اس کا syntax سمجھ لیں۔

```
scanf("format specifier", address);
```

()scanf فنکشن سے ان پٹ لیتے ہوئے ہمیں دو چیزیں بتانی پڑتی ہیں، ایک تو فارمیٹ یعنی ملنے والے ڈیٹا کو میموری میں کس فارمیٹ میں محفوظ کرنا ہے اور دوم میموری کا وہ ایڈریس جہاں پر ہم اس ڈیٹا کو محفوظ کروانا چاہتے ہیں۔
اب یہاں ایک مسئلہ آتا ہے۔ وہ یہ کہ Variables تو Run-Time میں Dynamically بنتے ہیں، اب ہمیں کس طرح پتا چلے گا کہ کون سا ویری ایبل میموری کی کس لوکیشن پر بنا ہے۔ ہمیں تو صرف ویری ایبل کا نام ہی پتا ہے۔ اس مسئلے سے بچنے کے لیے ہم استعمال کرتے ہیں ایڈریس آپریٹر۔

ایڈریس آپریٹر (&) کیا ہے؟

ایڈریس آپریٹر ++C کی زبان میں Ampersand کے سمبل (&) کو کہتے ہیں۔ اسے ہم جس ویری ایبل کے ساتھ لکھ دیتے ہیں یہ آپریٹر ہمیں اُس کا ایڈریس Return کرتا ہے۔ مزید کوئی نیا ٹاپک شروع کرنے سے پہلے ()scanf کو سمجھنے کے لیے کچھ مثالوں سے وضاحت کر لیتے ہیں:

مثال نمبر 1

ہم ایک پروگرام بنانا چاہ رہے ہیں جو ہم سے دو نمبرز ان پٹ لے اور دونوں کو جمع کرنے کے بعد ہمیں رزلٹ دکھادے

```
#include <conio.h>
#include <stdio.h>
void main(void)
{
int i, j;
clrscr();
printf("Enter 1st Number:");
scanf("%d",  &i);
printf("Enter 2nd Number:");
scanf("%d",  &j);
int sum;
sum= i + j;
printf("Result is= %d", sum);
getch();
}
```

پروگرام کو اچھی طرح پڑھے اور سمجھنے کی کوششیں کریں۔

سب سے پہلے تو ہم نے Include Directives دیئے ہیں جن سے آپ بخوبی واقف ہیں۔پہلی لائن کی مدد سے ہم نے ویری ایبل تخلیق کیے iاور j جن کی ڈیٹا ٹائپ Integer ہے۔

اسکرین کو Clear کروانے کے بعد ایک Message پرنٹ کروایا اور ویلیو ان پُٹ لی۔اسی طرح دوسرا ان پُٹ حاصل کیا۔

ایک نیا ویری ایبل تخلیق کیا جس کا نام sum رکھا۔i اور j دونوں کو جمع کر کے sum کو Assign کر دیا اور آخر میں sum کو پرنٹ کروادیا۔

getch() یوزر اسکرین کو روکے رکھنے کے لیے استعمال کیا۔

اب آپ مندرجہ بالا پروگرام کو دیکھتے ہوئے بہ آسانی تفریق،ضرب اور تقسیم کے پروگرامز بنا سکتے ہیں۔

آپریٹرز (Operators)

سادہ اور عام فہم الفاظ میں یوں سمجھ لیں کہ Operators کچھ Symbols یا گرافک تصویریں ہوتی ہیں جو کسی ویری ایبل کی ویلیو تبدیل کرنے میں یا کسی بھی ڈیٹا کو Manipulate کرنے میں پروگرام کی مدد کرتی ہیں۔

مزید آسان الفاظ استعمال کریں تو Operators بالکل فنکشن ہی ہوتے ہیں۔بس ہم نے کیا کیا کہ جمع کے فنکشن کا نام PLUS رکھنے کی بجائے + سمبل رکھ دیا جو روٹین plus کے لیے Call ہوتی وہی اس سمبل کے لیے Call ہو رہی ہے۔اس طرح سے پروگرام کو سمجھنا اور لکھنا آسان ہو گیا۔

ہم Operators کو Categorize کرتے ہیں دو طرح سے: کام کے اعتبار سے اور استعمال کے طریقہ کار کے لحاظ سے۔استعمال کے طریقہ کار کے لحاظ سے ہم Operators کو مندرجہ ذیل تین کیٹیگریز میں تقسیم کرتے ہیں۔

1-Unary operators
2- Binary operators
3- Ternary operators

UNARY OPERATORS

وہ آپریٹرز جن کے ساتھ صرف ایک Operand (وہ ویلیو یا ویری ایبل جس پر Operator اثر انداز ہو رہا ہے) ہو، Unary کہلاتے ہیں۔جیسے & آپریٹر۔

BINARY OPERATORS

وہ آپریٹرز جن کے ساتھ دو Operands ہوں جیسے + , - , * ,/ وغیرہ۔مثلاً A+B

TERNARY OPERATORS

عام طور پر لینگویجز میں Ternary آپریٹرز کا کوئی Concept ہوتا نہیں ہے۔ C++ میں ایک آپریٹر ہے ? مارک۔جسے ہم Conditional operator کہتے ہیں۔اسے ہم Ternary میں شمار کرتے ہیں کیونکہ اس کے ساتھ بیک وقت تین Operands استعمال ہوتے ہیں۔

کام کے اعتبار سے Operators کی قسمیں

1-Arithmetical Operators
2- Relational Operators
3-Logical Operators
4- Assignment Operators

5- Adderess Operators
6-Increment/Decrement Operators

اس کے علاوہ بھی ++C میں Operators کی بہت سی قسمیں ہیں جنہیں ہم وقتاً فوقتاً ڈسکس کرتے رہیں گے۔

ARITHMETICAL OPERATORS

وہ آپریٹرز جن کی مدد سے ہم ریاضی کے بنیادی مسائل حل کرتے ہیں اور مختلف Expressions تخلیق کرتے ہیں، ان آپریٹرز کی فہرست اور کام مندرجہ ذیل ہیں

جمع	**Addition**	+
تفریق	**Subtraction**	-
ضرب	**Multiplication**	*
تقسیم	**Divide**	/ **(Slash symbol)**
حاصل	**Modulus/ Remainder**	%

موضوع بہت خشک چل رہا ہے۔ چلئے میں آپ کو Operators کی مناسبت سے ایک شعر سنا دیتا ہوں۔

کاش میں خود کو ریاضی کے سوالوں کی طرح
تجھ پہ تقسیم کروں کچھ بھی نہ حاصل آئے

RELATIONAL OPERATORS

ان کی مدد سے ہم مختلف Comparision (موازنہ) کے expressions تخلیق کرتے ہیں۔ ان کی فہرست اور کام مندرجہ ذیل ہیں۔

< **Less than**
> **Greater than**
<= **less than or Equal**
>= **Greater than or Equal**
== **Equal**
!= **Not Equal**

LOGICAL OPERATORS

دو مختلف expresions کے باہم ملاپ یا جوڑنے کے لیے ہم عموماً اس طرح کے Operators کو استعمال کرتے ہیں۔ فہرست مندرجہ ذیل ہے:

&& **and**
|| **or**
! **not**

(=)ASSIGNMENT OPERATOR

یہ آپریٹر، ویری ایبل کو ویلیو Assign کرنے کے لیے استعمال ہوتا ہے۔ مثلاً A=5، A=i+j، A=10+5 وغیرہ۔

(&) ADDRESS OPERATOR

کسی بھی ویری ایبل کا ایڈریس معلوم کرنے کے لئے یہ آپریٹر اس کے ساتھ لکھا جاتا ہے۔

INCREMENT / DECREMENT OPERATOR

کسی ویری ایبل کی ویلیو میں ایک کا اضافہ کرنے کے لیے ہم Increment آپریٹر (++) اور ایک کی کمی کرنے کے لیے ہم Decrement آپریٹر (--) استعمال کرتے ہیں۔ یہ دونوں UNARY آپریٹرز کی مثالیں ہیں۔

++ Increment by 1

-- Decrement by 1

POSTFIX/PREFIX EXPRESSIONS

Increment یا Decrement آپریٹرز کو استعمال کرتے ہوئے اس بات کا خیال رکھیں کہ ان آپریٹرز کو ہم Operand یا ویری ایبل سے پہلے بھی لکھ سکتے ہیں اور ویری ایبل کے بعد بھی۔ اگر ہم آپریٹر پہلے لکھیں گے تو پہلے آپریٹر Apply ہوگا اور بعد میں Expression ویری ایبل کی ویلیو پڑھے گا جبکہ اگر آپریٹر کو ہم بعد میں Postfix کے طور پر استعمال کرتے ہیں تو Expression پہلے ویری ایبل کی ویلیو پڑھے گا اور بعد میں آپریٹر Apply کرے گا۔

مثال کے طور پر اگر ویری ایبل A پر یہ آپریٹرز Apply کرنے ہوں تو ہم مندرجہ ذیل طریقوں سے لکھ سکتے ہیں۔

1. A++ Postfix Increment
2. ++A Prefix Increment
3. --A Prefix Decrement
4. A-- Postfix Decrement

آ یئے Postfix اور Prefix کے فرق کو ایک مثال سے سمجھ لیتے ہیں۔

```
#include <conio.h>
#include <stdio.h>
void main(void)
{
int i, j;
i= j= 5;
printf(" i = % d\n" , ++i);
printf("i = % d\n", i++);
printf("i = %d\n " , i);
printf("j= %d\n" , j--);
printf("j= %d\n" , --j);
printf("j= %d\n" , j);
j++ ; ++j;
printf("j= %d\n" ,j);
getch();
}
```

مندرجہ بالا پروگرام آؤٹ پٹ (output) یہ آئے گا:

```
i = 6
i = 6
i= 7
j= 5
j= 3
j= 3
j= 5
```

پروگرام کو اچھی طرح پڑھنے سے آپ یقیناً Prefix اور Postfix کی منطق کو سمجھ جائیں گے۔ پروگرام کی دوسری لائن کو دیکھیں:

i = j= 5;

یہ لائن Multiple Assignment کہلاتی ہے۔ اس طرح کی کمانڈز میں ہم ویری ایبل کی ایک قطار لکھ دیتے ہیں اور آخری میں ایک ویلیو دے دیتے ہیں۔ اس طرح تمام ویری ایبل کو ایک ہی ویلیو Assign ہو جاتی ہے۔

Increment یا Decrement آپریٹرز کے حوالے سے ایک سوال عموماً سننے میں آتا ہے، وہ یہ کہ جب ہم ایک expression کی مدد سے باآسانی ویری ایبل کی ویلیو میں 1 کا اضافہ کر سکتے ہیں تو Increment یا Decrement آپریٹرز کو کیوں استعمال کریں؟ ایک ہی لینگویج میں ایک ہی کام کے لیے دو چیزیں کیوں؟ چلیں مان لیا کہ ++، Syntax کے اعتبار سے لکھنے اور سمجھنے میں آسان ہے مگر ہم پروگرامنگ کرتے ہوئے دونوں طریقوں میں سے کون سا اختیار کریں اور کیوں۔

مثال کے طور پر A ایک ویری ایبل ہے، جس میں ہم 1 کا اضافہ کرنا چاہ رہے ہیں۔ اب اس کے مندرجہ ذیل دو طریقے ہیں

1. A++ ; یا ++A ;

2. A = A + 1 ;

اب دونوں صورتوں میں A میں ایک کا اضافہ ہو جائے گا، ہم کس کو کس پر برتری دیں اور کیوں؟

آسان الفاظ میں اس کا جواب یہ ہے کہ ویری ایبل کا نام ایک مرتبہ Declare یا تخلیق ہونے کے بعد پروگرام میں جہاں کہیں بھی آتا ہے، Compiler اُس جگہ میموری کو پڑھتا ہے اور ویری ایبل سے مراد اُس کی ویلیو سمجھتا ہے۔ اگر A ایک ویری ایبل ہے اور اس کی ویلیو 5 ہے تو A کو پرنٹ کروانے پر جواب 5 ہی آئے گا، A نہیں۔

اتنا تو آپ جانتے ہی ہوں گے کہ میموری کی اسپیڈ (Speed) پروسیسر کے مقابلے میں کم ہوتی ہے۔ لہٰذا A=A+1; میں دو بار A استعمال ہوا ہے۔ کمپائیلر کو دو مرتبہ میموری کو Read کرنا ہوگا، چنانچہ A++ میں پروگرام کی رفتار زیادہ ہوگی۔ اسی طرح

```
A=A+1;
C=A+B;
B=B+1;
```

کو ہم اس طرح بھی لکھ سکتے ہیں۔

C = ++A + B++ ;

اب پہلے والے Code میں میموری 7 مرتبہ Read کی گئی جبکہ دوسرے code میں صرف 3 بار۔

اسی لیے پروگرامنگ میں ہم ترجیح دیتے ہیں Increment اور Decrement آپریٹرز کے استعمال کو۔

Operators کی ہی مناسبت سے ایک بات اور سمجھ لیں۔ Arithmetic operators کو اگر ہم اسائنمنٹ آپریٹر سے پہلے لکھ دیں تو اس کا مطلب ہوتا ہے کہ ٹارگٹ ویری ایبل، Source ویلیو یا ویری ایبل کے ساتھ Operation میں شامل ہے۔ مثلاً A =A+1; کو ہم یوں بھی لکھ سکتے ہیں:

A += 1;

دونوں Expressions کا مطلب یکساں ہے۔ ان آپریٹرز کی فہرست درج ذیل ہے۔ یہ آپریٹرز Arithmetic Assignment آپریٹرز کہلاتے ہیں:

1. +=
2. -=
3. *=
4. /=
5. %=

آپریٹرز Operation کے وقت ریاضی کے مطابق ہی BODMAS اصول کی پیروی کرتے ہیں۔ مزید آپ کو ان کی ترتیب یا Precedence، C++ کی help میں مل جائے گی۔

help دیکھنے کا طریقہ

آپ Alt+H کی مدد سے Help کے مینو پر جائیں یا F1 پریس کر کے مطلوبہ ٹاپک کو دیکھ لیں یا پھر سب سے آسان طریقہ یہ ہے کہ آپ جس چیز سے متعلق Help چاہتے ہیں اُس کا نام Editor میں لکھیں اور نام کے نیچے کسی بھی جگہ پر کرسر لا کر Ctrl+F1 پریس کر دیں۔ Help آپ کے سامنے ہوگی۔ مثال کے طور پر آپ clrscr() سے متعلق Help چاہتے ہیں تو آپ Editor کی فائل میں clrscr() لکھ کر، کرسر اس کے نیچے کسی بھی کیریکٹر پر لے آئیں اور Ctrl+F1 پریس کر دیں۔ Help آپ کے سامنے موجود ہوگی۔ چلیں! چند مثالوں کے ساتھ Operators کے ٹاپک کو ختم کرتے ہیں۔ بتائیے مندرجہ ذیل Expressions کے بعد X کی ویلیو کیا ہوگی:

Problem # 01

```
int x;
x= ( 3 + 5) * 2 + 2;
x = ?
```

جواب آئے گا 18۔ سب سے پہلے parenthesis حل ہوں گے۔ پھر 8 ضرب ہوگا 2 کے ساتھ اور آخر میں 16 جمع 2، یعنی 18 آجائے گا۔

Problem # 2

```
int a = 5;
a += (a +5);
a = ?
```

جواب آئے گا 15۔ سب سے پہلے Parenthesis حل ہوں گے یعنی a += (5+5)

a +=10

اور a +=10 برابر ہے a= a + 10 کے یعنی a = 5+10 تو جواب آیا 15۔ ایک اور پرابلم دیکھئے:

Problem # 03

```
int x =5;
int y = -10;
int z =0;
z = x-y;
z=?
```

جواب آئے گا -5 y کی ویلیو ہم نے -ve کر دی۔ y پہلے ہی منفی میں تھا لہٰذا مائنس، مائنس + ہو گئے۔ Expression اب ہو گیا:

```
z = x - 10
z = 5 + 10
z= 15
```

ایک اور پرابلم دیکھئے

Problem #04

```
int x = 5; y = -10 ; z=0;
z= x++  +  ++y;
z = ?
```

جواب آئے گا -4۔ جب y میں 1 کا اضافہ ہوگا تو ویلیو ہو جائے گی -9، جب -9 میں 5 جمع ہوں گے تو ویلیو بچے گی -4۔ کیونکہ x پر ++ بعد میں Apply ہوگا۔

امید ہے مندرجہ بالا مثالوں سے آپ بخوبی Operator کے کام کرنے کا طریقہ کار سیکھ چکے ہوں گے۔

باب نمبر 6

Decision Statements

Decision Making کیا ہے؟

Decision Making یا فیصلہ کرنے کی صلاحیت حضرت انسان کی وہ خوبی ہے جس کی وجہ سے اُسے Computer پر برتری حاصل ہے۔ حضرت انسان مختلف حالات میں فیصلے کرنے اور اُن پر عمل کرنے کی صلاحیت رکھتے ہیں جبکہ کمپیوٹر، جیسا کہ ہم پہلے بھی ذکر کر چکے ہیں، ایک بے وقوف مشین، حکم کا غلام یا لکیر کا فقیر ہے۔ جو بتا دیا سو کر لیا ورنہ ہاتھ پر ہاتھ دھر کر بیٹھے ہیں، خود سے کچھ بھی نہیں کریں گے۔ نہ اچھا نہ بُرا۔

اب انسانوں نے سوچا کہ کیوں نہ کمپیوٹر بھی سوچے، مختلف صورت حال میں مختلف کام کرے، کبھی Output کچھ آئے تو کبھی کچھ۔ بھلے یہ سوچ بہت مختصر ہی یا محدود ہی کیوں نہ ہو۔ اب وہ کمانڈز یا ہدایات جن کی مدد سے ہم کمپیوٹر کو سوچنے کا طریقہ یا کمپیوٹر پروگرام کو Thinking Criteria بتاتے ہیں، یا وہ ہدایات جو Decision Making والی خصوصیت کی Implementations میں ہماری مدد کرتی ہیں، یا جن کی مدد سے ہم پروگرام کو سوچنے کے لیے کہتے ہیں وہ Decision making کہلاتی ہیں۔

C++ میں ایسے 3 اسٹرکچرز موجود ہیں:

1- if

2- switch

3- Conditional operator?

IF

سب سے پہلے Syntax کو سمجھ لیں۔

```
if (condition)
do this;
```

ہم if کے بعد بریکٹس میں کوئی condition لکھ دیتے ہیں۔ اگر کنڈیشن True (صحیح) ہوگی تو بعد والی statement کام کرے گی ورنہ کمپائلر اُس Statment کو execute نہیں کرے گا۔

یاد رکھیں! C++ میں صفر (Zero) سے مراد False ہے اور صفر کے علاوہ کوئی بھی نمبر خواہ وہ پوزیٹو (+ve) ہو یا نیگیٹو (-ve) وہ True ہی شمار ہوگا۔

if else

if else structure ہم استعمال کرتے ہیں دونوں صورتوں کے لیے۔ اگر کنڈیشن True ہو تو یہ کرو ورنہ یہ کرو۔

```
Syntax:
if (Condition) do this;
else do this;
```

آیئے! پروگرام کی مدد سے سمجھ لیتے ہیں۔

ہم چاہ رہے ہیں کہ پروگرام ہم سے ایک نمبر ان پٹ لے۔ اگر نمبر 10 کے برابر یا اس سے چھوٹا ہو تو لکھا ہوا آئے Yes! اور اگر نمبر 10 سے بڑا ہو تو لکھا ہوا آئے Invalid

PROGRAM:

```
#include < conio.h>
#include < stdio.h>
void main(void)
{
int i ;
clrscr();
printf("Enter Any number :");
scanf("%d",  &i);
if (i <=10)
printf("YES!");
else
printf("Invalid!");
getch();
}
```

امید ہے آپ پروگرام سمجھ گئے ہوں گے۔ آیئے ایک اور پروگرام بناتے ہیں۔

ہم چاہ رہے ہیں کہ پروگرام ہم سے ایک نمبر ان پُٹ لے۔
اگر نمبر ODD یعنی طاق ہو تو اسکرین پر لکھا ہوا آ جائے ODD اور اگر نمبر جُفت (Even) ہو تو لکھا ہوا آ جائے Even

```
# include <conio.h>
# include <stdio.h>
void main(void)
{
int i;
clrscr();
printf("Enter Any no:");
scanf("% d" , &i);
if (i % 2 == 0)
printf("Even !");
else  printf("ODD!");
getch();
}
```

دیکھئے! ہم نے % ماڈیولس یا Remainder آپریٹر Apply کیا ہے۔ ملنے والا نمبر 2 سے تقسیم ہوگا۔ اگر حاصل صفر آئے گا تو مطلب ہوا نمبر Even ہے ورنہ ODD ہے۔ اب آپ ایسا کیجئے کہ مندرجہ بالا پروگرام کو Re-write کیجئے مگر == برابر کا سائن اور صفر کو استعمال نہ کریں۔ اس طرح لکھیں گے:

```
if (i%2)
printf("ODD!");
else
printf("Even!");
```

صرف messages کو اُلٹا کر دیا اب بھی پروگرام درست ہے۔ دیکھیں کس طرح: اگر Remainder صفر آیا تو صفر کا مطلب C++ میں ہوتا ہے False اور False کی صورت میں else، Execute ہوگا اور اسکرین پر Even لکھا ہوا آ جائے گا جبکہ صفر کے علاوہ کوئی ویلیو آتی ہے تو اس کا مطلب ہوا True لہٰذا ODD لکھا ہوا آ جائے گا۔ ہے نا مزے کی بات! چلیں! اب آپ جلدی سے مندرجہ ذیل پرابلم کا رزلٹ بتایئے

Problem:

```
int  i=4;
if (i = (10+5))  i++ ;
i = ?
```

جواب آئے گا 16......کیوں؟

بھئی! آسان سی بات ہے۔ بریکٹس میں ہم نے == ڈبل Equal سائن یعنی Comparision استعمال نہیں کیے بلکہ اسائنمنٹ آپریٹر استعمال کیا ہے۔ i کو 10+5 15, مل جائے گا۔ C++ میں 15 سے مراد True ہے (صفر کے علاوہ کوئی بھی نمبر True شمار ہوتا ہے) لہٰذا i میں ایک کا اضافہ اور ہو جائے گا اور جواب 16 آئے گا۔

Swaping

ہم چاہ رہے ہیں کہ ایک پروگرام بنائیں جو دو ان پُٹ ہم سے لے اور پھر دونوں ویری ایبلز کی ویلیوز کو آپس میں تبدیل یا Interchange کر دے۔ کمپیوٹر پروگرامنگ میں یہ طریقہ کار Swapping کہلاتا ہے۔

Program:

```
#include < conio.h>
#include < stdio.h>
void main( void)
{
int a, b, temp;
printf("enter ist velue:");
scanf("%d",  &a);
printf("Enter 2nd value:");
scanf("%d",  &b);
temp = a;
a=b;
b=temp;
printf("values after swapping:");
printf("\n%d\n",a);
printf("%d",  b);
getch();
}
```

اسی پروگرام کو دوبارہ لکھیں، **Swaping** کروائیں مگر کوئی تیسرا ویری ایبل استعمال کیے بغیر۔ دیکھیے! اس طرح

```
a= a+b;
b= a -b;
a= a-b;
```

NESTED IF

اگر ہم ایک **IF** باڈی کے اندر ایک اور **IF** لکھتے ہیں یا **else** باڈی میں ایک اور **IF** لکھتے ہیں تو اس کو ہم **IF** کی **Nesting** کہتے ہیں۔ یہاں ایک بات اور سمجھ لیں وہ یہ کہ **IF Statement** کے بعد جو کچھ بھی ہم لکھتے ہیں، **else** سے پہلے تک وہ **If Body** اور **else** کے بعد والی **Statement** کو **elseBody** کہا جاتا ہے۔ اب چاہے وہ ایک **Statement** ہو یا بہت سی **statements** کا مجموعہ ہو۔

Multiple statements کیلئے ہم کرلی بریکٹس {} کے درمیان **statement** کو لکھتے ہیں۔ کیونکہ **IF** یا **else** صرف اور صرف ایک **statement** کو **condition** کے **True** یا **False** ہونے کی صورت میں **Execute** کرتا ہے۔ خواہ وہ **Statement** ' **Simple** ہو یا **Compound**۔

چلیں **Nested** کی منطق کو مندرجہ ذیل پروگرام سے سمجھتے ہیں:

```
#include <conio.h>
#include <stdio.h>
void main(void)
{
int i;
printf( "Enter either 1 or 2") i;
scanf(" %d" ,&i);
if (i = = 1)  printf(" win");
else
   {
   if (i = = 2) printf("loss") ;
   else printf ("Invalid!") ;
    }
}
```

مندرجہ بالا پروگرام کو دیکھیے، اس میں ہم نے **if- else construct** کو **Nested** کیا ہے۔

if- else میں پہلے **if** کی **body** ہے۔ پھر **else** میں ایک اور **if** لکھا ہے۔

اگر پہلی کنڈیشن **FALSE** ہوگی تو دوسری **condition** چیک ہوگی۔ دوسری **condition** بھی **FALSE** ہوگئی تو دوسرا **ELSE** ایگزیکیوٹ ہوجائے گا۔

if-else میں **Nesting** کی کوئی حد نہیں، آپ جتنے چاہیں اتنے لیول تک **Nesting** کر سکتے ہیں۔ ہاں! اگر آپ صرف اور صرف بہت سارے **IF** لکھ دیتے ہیں اور **ELSE** کا استعمال نہیں کرتے تو کمپائیلر تمام **conditions** کو **Sequentially** پڑھے گا۔ پہلی **Condition** کے **True** ہوجانے کے باوجود دوسری تمام **Conditions** کو چیک کرے گا۔ اس کے برعکس اگر آپ **else** ساتھ ساتھ لگاتے جائیں گے تو **IF** کے **True** ہوجانے کی صورت میں کمپائیلر مکمل طور پر **else** والے حصے کو **Ignore** کردے گا۔ جس سے پروگرام کی **Execution Speed** بڑھے گی۔

DANGLING ELSE

Dangling Else کی اصطلاح ہم تب استعمال کرتے ہیں جب پروگرام میں **if** کی تعداد کے مقابلے میں **else** کی تعداد کم ہو اور یہ سمجھنا مشکل ہو رہا ہو کہ **else** کونسے **if** کے ساتھ ہے۔ چلیں ایک پروگرام سمجھتے ہیں:

```
#include <conio.h>
#include <stdio.h>
void main (void)
{
int i = 3;
if (i > 100)
   if (i > 50)
      if (i > 5)
        pri ntf(" > f ive ");
     else  printf(" < 100");
getch() ;
}
```

جی جناب! بتایئے، پروگرام کا **Output** کیا آئے گا؟

اس پروگرام کا **output** کچھ نہیں آئے گا! کیونکہ **C++** میں صرف کالم کی **Settings** سے کچھ فرق نہیں پڑتا، آخر میں لکھا ہوا **else** آخری **if** کا ہے۔

ELSE JOINING

else کی Joining کا طریقہ کار یہ ہے کہ else اپنے سے پہلے آنے والے قریب ترین If کے ساتھ Join ہو جاتا ہے۔
مگر اس قریب ترین If کی Searching میں Curly Brackets (کرلی بریکٹس) کو Ignore کر دیتا ہے۔

مثلاً آپ مندرجہ بالا پروگرام دوبارہ دیکھنا چاہ رہے ہیں مگر آپ چاہتے ہیں کہ اس کا آؤٹ پٹ Less than 100 آئے تو کیا کریں گے۔ Else تو جب بھی لکھیں گے وہ آخری والے If کے ساتھ جا ملے گا۔ اس طرح کی کیفیت میں ہم کرلی بریکٹس کا استعمال کرتے ہیں۔ آپ صرف اتنا کریں کہ پہلے والے If کے بعد { Opening بریکٹ اور Else سے پہلے } Closing بریکٹ لکھ دیں۔ اب Else جب اپنے سے پہلے آنے والے If کو تلاش کرے گا تو کرلی بریکٹس کے مکمل Pair کو Ignore کر دے گا۔ اس طرح وہ پہلے والے If کے ساتھ join ہو جائے گا۔

اس طریقے پر عمل کرتے ہوئے آپ Else کو دوسرے If کے ساتھ بھی ملا سکتے ہیں۔ آپ کو صرف اس منطق کو سمجھنے کی ضرورت ہے۔
If کے Topic کو ختم کرنے سے پہلے ایک بات اور سمجھ لیں۔ وہ یہ کہ یوں تو If کی کوئی Limit نہیں ہے۔ آپ ایک ہزار If استعمال کریں، کمپائلر ہرگز ہرگز آپ کو کچھ نہیں کہے گا۔ مگر سافٹ ویئر معیارات (Standard) کے تحت ہم If-else کو صرف تین لیولز Depth تک استعمال کرتے ہیں۔
If Statement کی مندرجہ ذیل ممکنہ صورتیں ہو سکتی ہیں:

```
1) if (condition)
      do this;
```

```
2) if (condition)
    {
     do this;
     and this;
    }
```

```
3) if (condition)
      do this;
   else
      do this;
```

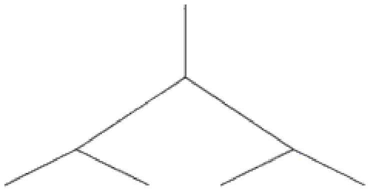
```
4) if (condition)
    {
     do this;
     and this;
    }
    else
     {
       do this;
       and this;
     }
```

```
5) if (condition)
     do this;
   else
     {
      if (condition)
        do this;
      else
        {
         do this;
         and this;
        }
     }
```

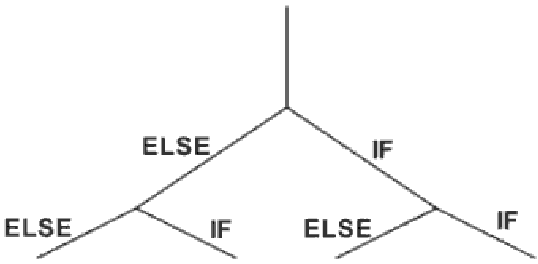
```
6) if (condition)
    {
      if (condition)
        do this;
      else
       {
        do this;
        and this;
       }
    }
    else
       do this;
```

Decision Tree

If-Else کو ہم ایک Tree کی مدد سے بھی ظاہر کر سکتے ہیں۔ Decision Tree ہمیں بتاتا ہے کہ پروگرام میں کتنے If اور کتنے Else استعمال ہوئے ہیں اور ان کی ترتیب کیا ہے۔ یاد رکھئے کہ کمپائلر Decision Tree کو ہمیشہ Right سے پڑھنا شروع کرے گا اور پھر left پر جمپ کرے گا۔ Tree میں موجود تمام دائیں شاخ If کو اور تمام بائیں شاخ Else کو Show کرتے ہیں۔ چلیں مثال سے سمجھتے ہیں۔ مندرجہ ذیل Decision Tree کو حل کیجئے:



حل:



مندرجہ بالا Tree میں تین عدد If اور 3 ہی Else ہیں۔ ان کی ترتیب یہ ہوگی۔

```
if
 {
  if
  else
 }
else
 {
  if
  else
 }
```

چلیں، جناب If تو واضح ہوا۔ اب بات ہو جائے کچھ Conditional آپریٹر پر۔

Conditional Operator ?

? کنڈیشنل آپریٹر If-Else کی مشترکہ شکل ہے۔ مثال کے طور پر ہم یوں لکھتے ہیں۔

```
if (x > 5)
  y = 3;
else
  y = 4;
```

اس Expression کو ہم ? آپریٹر کی مدد سے یوں بھی لکھ سکتے ہیں۔

```
y = (x > 5 ? 3 : 4)
```

اب اگر کنڈیشن، True ہوگی تو پہلی Value اور اگر False ہوگی تو (:) کولون سائن کے بعد والی ویلیو y کو مل جائے گی۔ یہ ضروری نہیں کہ ہم ? آپریٹر کو صرف نیومیرک Calculation میں ہی استعمال کریں۔ مثال کے طور پر:

```
printf("Enter any Character:");
scanf("%c",  &a);
y = (a >= 65 && a  <= 90 ? 1 : 0);
```

ان نیومیرک Calculation کے علاوہ بھی ہم ? آپریٹر کو استعمال کر سکتے ہیں۔ مثال کے طور پر

```
char a = 'z' ;
printf("%c", (a >= 'a' ? a : '!'));
```

ایک اور مثال دیکھئے:

```
(i==1 ? printf("Yes! ") : printf("No"));
```

Conditional آپریٹر عموماً تب استعمال کیا جاتا ہے جب ہم If-Body اور Else-Body میں صرف ایک Statement دینا چاہیں۔ Multiple-Statement کے لئے ہم If-Else کو ہی استعمال کرتے ہیں۔

Switch

ایسی Statements جہاں ہم کو Choice کرنی ہو بہت سے Cases پر، مثال کے طور پر عام زندگی میں کس کالج میں داخلہ لیا جائے، کون سا Major چنا جائے، کس ہوٹل میں قیام کیا جائے۔ ہمارے پاس بہت سارے راستے اور طریقہ کار ہوتے ہیں جن میں سے ہمیں ایک کا انتخاب کرنا ہوتا ہے۔ اب اسی Situation کو ہم Implement کرنا چاہ رہے ہیں C++ میں۔ ایک طریقہ تو یہ ہے کہ ہم If کو قطار در قطار لکھتے چلے جائیں، یا ہم Swicth کو استعمال کریں۔ Swicth ہم استعمال کرتے ہیں جب ہم بہت سی Conditions کو چیک کرنا چاہ رہے ہوں۔ یا آسان الفاظ میں بہت سے راستوں یا طریقہ کار میں ہمیں مختلف Output چاہئے۔

Syntax:

```
switch(Interger/char variable)
{
case constant 1 : do this; break;
case constant 2  : do this; break;
:
case constant n : do this; break;
default : do this;
}
```

Switch کے ساتھ ہم وہ Test Variable دیتے ہیں جس کی ویلیو ہم چیک کرنا چاہ رہے ہیں۔ Case، کی ورڈ Keyword ہے جس کے بعد ہم ایک Constant ویلیو دیتے ہیں جو کہ Variable کی ویلیو کے ساتھ Match ہوتی ہے۔ اگر دونوں Same ہوں گی تو اس کا مطلب ہے کہ True Condition ہے اور اس کے بعد Compiler کولون سائن (:) کے بعد لکھی گئی ہدایات پر عمل کرے گا۔ خواہ وہ ایک ہو یا بہت سی۔ ہر Case کے بعد Break اس لئے لگاتے ہیں تا کہ کمپائیلر Case کو Exemple کرنے کے بعد کنٹرول، Switch کے بعد والی Statement پر ٹرانسفر کر دے۔
اگر آپ Break نہیں دیں گے تو True Case کے بعد والے تمام Cases، خود بخود Run ہو جائیں گے۔ اب ظاہری بات ہے کہ اگر ایک کیس True ہوگا تو اسی وقت میں کوئی دوسرا True نہیں ہو سکتا۔

Default

Default کی ورڈ (Keyword) بالکل Else والا کام کرتا ہے کہ اگر کوئی بھی کیس Match نہ ہو تو default والی Statments پر عمل ہو۔ default کے بعد ہم Break اس لئے استعمال نہیں کرتے کیونکہ default تو ہمیشہ آخر میں ہی ہوتا ہے اور اس کے بعد کمپائیلر خود بخود Switch کے بعد والی Statement پر ہی عمل کرے گا تو یہاں پر Break بے معنی ہو جاتا ہے۔ آیئے! ایک مثال سے Switch کے کام کرنے کے طریقہ کار کو سمجھتے ہیں۔

```
# include <conio.h>
# include <stdio.h>
void main(void)
{
int i=2;
switch(i)
{
case  1 : printf("I am in one");
 break;
case  2 : printf("I am in two");
 break;
case  3 : printf("I am in three");
break;
default  : printf("I am invalid!");
}
}
```

پروگرام کا Output آئے گا:

I am in two

کیونکہ دوسرے Case کا Constant، Variable کی Value کے ساتھ Match ہو گیا۔ default بھی Else کی طرح Optional ہے، آپ چاہیں تو استعمال کریں، چاہے تو نہ کریں۔ البتہ، Switch کو استعمال کرتے ہوئے کچھ باتوں کا خیال رکھئے:
یہ ضروری نہیں کہ constant ترتیب صعودی (Ascending Order) میں ہی دیں۔ آپ جس طرح چاہیں لکھ سکتے ہیں۔ مثال کے طور پر:

```
Int   i=2;
Switch (i)
{
Case   121 : Print ("I'm in 121);
Break
Case  2 : Print ("I'm in 2");
Break;
Case 7 : Print ("I'm in 7");
Break
```

اس پروگرام کا Output آئے گا۔

I'm in 2

Switch کے ساتھ آپ Integer، ویری ایبل کے علاوہ وہ کیریکٹر بھی استعمال کر سکتے ہیں۔ مثلاً

```
Char C = 'x' ;
Switch (C)
{
Case 'y' : Input ("yes!"); Break:
Case 'n' : Print ("No!"); Break;
Default : Print ("Invalid!");
```

اس پروگرام کا Output آئے گا۔

Invalid

کچھ حالات میں ہم Case کے بعد کوئی Statement نہیں لکھتے۔ جیسے ہی کیس True ہوگا، وہ Break تک تمام Statement کو Execute کرے گا۔ مثلاً

```
Char  A = 'C' ;
Switch (A)
{
Case 'C' ;
Case  'C' ;
Print (Applicat C") ;
Break ;
```

اس پروگرام میں C Lowercase میں ہو یا Uppercase میں Output، یکساں رہے گا۔

Case میں ایک اسٹیٹمنٹ ہو یا Multiple، دونوں صورتوں میں کرلی بریکٹس استعمال نہیں ہوں گے، کولن سمبل اور Break ایک مکمل Pair کا ہی کام کرتے ہیں۔

اگر ہمیں Typical Conditions استعمال کرنی ہوں تو ہم Switch پر If کو فوقیت دیتے ہیں کیونکہ Case میں ہم یوں نہیں لکھ سکتے۔ مثلاً

```
Case  I < = 20
```

ان صورتوں میں ہم If کو ہی استعمال کریں گے۔

باب نمبر 7

LOOPS

اس باب میں ہم بات کریں گے لوپنگ اسٹرکچرز پر۔ LOOP کیا ہے؟ اس کی کتنی اقسام ہیں اور استعمال کا طریقہ کیا ہے؟ ان تمام باتوں کو ہم اس باب میں زیرِ بحث لائیں گے۔

LOOP کسے کہتے ہیں؟

LOOP کی معیاری تعریف **(Standard definition)** کچھ یوں ہے:

''ایک ہی کام کو، ایک سے زائد مرتبہ دہرانے کا نام **LOOP** ہے۔''

"Repetition of same task more than one time at a continuous level"

روزمرہ زندگی میں استعمال

روزمرہ زندگی میں ہم **Loop** کی کئی مثالیں دیکھتے ہیں۔ مثلاً تقریر میں ایک ہی بات کا بار بار دہرانا، سانس کا عمل ایک خاص تسلسل کے ساتھ، خون کی گردش، زمین کی محوری گردش، ستاروں اور پلکوں کا جھپکنا، پہیے کا گھومنا اور جھولے کا جھولنا اس قسم کی لاتعداد مثالیں ہیں جن سے ہمارا واسطہ روزمرہ زندگی میں پڑتا ہے۔

LOOP کی قسمیں

عام طور پر **LOOP** کی دو قسمیں ہوتی ہیں:

1- وہ **LOOP** جو کسی **Condition** کے تحت چلیں۔ **Condition** کے **True** یا **False** ہونے کی صورت ان کے چلنے یا رکنے کی نشاندہی ہو جائے۔

2- وہ **LOOP** جو مقررہ تعداد تک چلیں۔ مثال کے طور پر **10** مرتبہ یا **20** مرتبہ۔ اس **Loop** میں ہم ایک نمبر بتاتے ہیں کہ **LOOP** کو یہاں سے گننا شروع کرنا ہے۔ یہ **LOOP Counter** کی ابتدائی قیمت ہوتی ہے اور ایک نمبر ہم اس کی انتہائی قیمت کے طور پر بتاتے ہیں۔

اگر ہم کام کرنے کے طریقہ کار کے لحاظ سے دیکھیں تو **LOOPS** کی دونوں قسموں (یعنی کنڈیشنل اور ان کنڈیشنل) میں **Condition** چیک ہوتی ہے۔ دونوں طرح کے **LOOP** کا فلوچارٹ ایک ہی بنے گا، مگر اول الذکر میں ہمیں یہ نہیں پتا ہوتا کہ **LOOP** کتنی مرتبہ چلے گا جبکہ ثانی الذکر میں ہمیں یہ بات بخوبی پتا ہوتی ہے۔

un-conditional لوپ میں **Counter** کی ویلیو بھی **Condition** کے ذریعے ہی چیک ہو رہی ہوتی ہے۔

اگر ہم ایک بڑے سارے پروگرام میں **Clrscr** کی کمانڈ مختلف جگہوں پر **10** بار استعمال کرتے ہیں تو کیا ہم اس کو **Loop** کہیں گے؟ ہرگز نہیں! مگر کیوں؟ یہ بھی تو ایک ہی کام کا بار بار انجام دینا ہے، مگر یہاں ایک ہی ساتھ **(At continous level)** والی شرط نہیں ہے۔

ایسا کرتے ہیں کہ پروگرام میں ایک ساتھ **clrscr**، **10** بار لکھ دیتے ہیں، اب کیا یہ لوپ ہے؟ جی ہاں! اسے ہم **Loop** کہیں گے کیونکہ ایک ہی **TASK** ایک ساتھ، ایک سے زائد مرتبہ انجام دے رہا ہے۔

ایک بات کو اچھی طرح ذہن میں بٹھا لیجئے، وہ یہ کہ **LOOP** ایک **Concept** کا نام ہے یہ کسی کمانڈ یا اسٹرکچر کا نام نہیں ہے۔

یہ **Concept** خواہ کسی بھی طرح پورا ہو رہا ہو ہم اسے **LOOP** کہیں گے۔

جو کمانڈز، **Statements** یا **Structures** ہمیں اس **Concept** کو **Implement** کرنے میں مدد دیتے ہیں یا دوسرے لفظوں میں جن **Statements** کی مدد سے ہم باآسانی **LOOP** کو **Implement** کر سکتے ہیں ان کو ہم **Looping structures** کہتے ہیں۔

اس طرح کے **3** تین اسٹرکچرز **C++** میں موجود ہیں۔ آیئے تینوں کو باری باری سمجھ لیتے ہیں۔

1- while loop

2-do.....while loop

3-for loop

while loop

while loop کے اندر ہم پہلے **Condition** لکھتے ہیں پھر وہ **Statement(s)** جو ہم بار بار **Execute** کروانا چاہتے ہیں۔

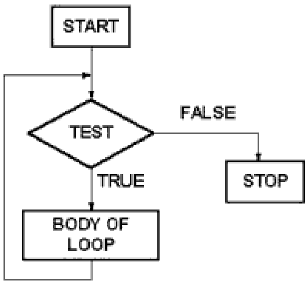
Syntax

```
while (condition)
do-this;
```

اگر ایک سے زائد Statements کو execute کروانا ہو تو ہم کرلی بریکٹس میں ایک سے زائد Statements کو قید کردیتے ہیں۔

```
while (Condition)
{
do-this;
and this;
and this;
}
```

while loop کے کام کرنے کے طریقہ کار کو آپ فلو چارٹ سے بخوبی سمجھ سکتے ہیں:



اگر کنڈیشن True ہوگی تو LOOP چلتا رہے گا۔ یہاں تک کہ کنڈیشن FALSE نہ ہو جائے۔

آیئے While کو پروگرام کی مدد سے سمجھ لیتے ہیں: ہم چاہ رہے ہیں کہ ایک پروگرام بنائیں جو کہ ہم سے ایک کے بعد ایک کیریکٹر ان پٹ لیتا رہے جب تک کہ ہم انٹر کی (Enter Key) پریس نہ کردیں:

(1)

```
#include <conio.h>
#include <stdio.h>
void main(void)
{
char ch;
while (ch !=`\r`)
{
ch=getch();
printf("%c",ch);
}
}
```

اب یہ پروگرام اس وقت تک ہم سے کیریکٹر input لیتا رہے گا جب تک کہ ہم انٹر پریس نہ کردیں۔

escape sequences میں آپ یہ تو پڑھ ہی چکے ہیں کہ انٹر کی کا کوڈ ' \r' ہے۔

اب مثال کے طور پر ہم A پریس کرتے ہیں تو while کی condition یوں چیک ہوگی۔

A ! = '\r' (enter A کے برابر نہیں ہے)

جواب آئے گا True اور loop پھر چل جائے گا۔ جب ہم enter پریس کردیں گے تو ch ویری ایبل کی ویلیو ہو جائے گی \r، اب condition کچھ یوں ہوگی:

`\r` !=`\r`

اور جواب آئے گا FALSE کیونکہ دونوں ویلیوز ایک دوسرے کے برابر ہیں۔ اس طرح while کا False,loop پر ختم ہو جائے گا اور کنٹرول Loop کے بعد والی لائن پر ٹرانسفر ہو جائے گا۔

getch کی مدد سے ہم ایک کیریکٹر get کرتے ہیں keyboard سے۔ مگر وہ کیریکٹر اسکرین پر پرنٹ نہیں ہوتا ہے۔ اگر ہم getche() استعمال کریں تو اس کا مطلب ہے get character with echo یعنی جو بھی کیریکٹر ہم پریس کریں گے وہ اسکرین پر بھی پرنٹ ہوتا چلا جائے گا۔

آیئے مندرجہ بالا پروگرام کو مختلف Logics سے بناتے ہیں۔

(2)

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
	char ch;
	while(ch !=`\r`)
		ch=getche();
}
```

(3)

```
#include <conio.h>
#include <stdio.h>
void main(void)
{
char ch;
while((ch=getche()) !=`\r`);
}
```

(4)

```
#include <conio.h>
#include <stdio.h>
void main(void)
{
while(getche() !=`\r`);
}
```

مندرجہ بالا چاروں پروگراموں کا آؤٹ پٹ یکساں آئے گا۔ امید ہے آپ سمجھ گئے ہوں گے۔

چلئے اب آپ جلدی سے ایک پروگرام بنایئے جو کہ ہم سے کیریکٹرز **input** لے اور جب ہم انٹر کی پریس کر دیں تو ہمیں بتائے کہ ہم نے کل کتنے کیریکٹرز **input** دیے ہیں۔

```
#include <conio.h>
#include <stdio.h>
void main(void)
{
int counter=0;
while(getche() !=`\r`)
        counter ++;
printf("Total Characters = %d", counter);
getch();
}
```

خلاصہ

while loop سے متعلق چند باتیں نوٹ کر لیجئے:

while-1 لوپ **true** پر چلتا ہے اور **FALSE** پر رک جاتا ہے۔

2- جو **condition** ہم **while** میں لکھتے ہیں وہ آسان بھی ہو سکتی ہے اور پیچیدہ بھی۔ ہم **Relational** آپریٹرز بھی استعمال کر سکتے ہیں اور **logical** بھی۔ مثلاً

```
while (i<=10)
while (i<=10 && j<=15)
while(j>10 && (i<15 || c<15))
```

3- **loop** کے ساتھ لکھی جانے والی اسٹیٹمنٹ ایک بھی ہو سکتی ہے اور ایک سے زائد بھی۔
ایک سے زیادہ ہونے کی صورت میں آپ کرلی بریکٹس کو استعمال کریں گے۔

4- اگر **condition** ہمیشہ **true** ہی رہے گی تو **infinite loop** چلتا رہے گا جیسے

```
while(i<=10)
   printf("%d",i);
```

اس **Loop** میں ہم نے **i** پر **increment** ہی نہیں لگایا لہٰذا کنڈیشن ہمیشہ **true** رہے گی۔

5- اگر آپ نے **Loop** کے فوراً بعد سیمی کولن لگا دیا تو **infinite loop** چلتا رہے گا۔ مثلاً:

```
while(i<=10);
i++;
```

6- ضروری نہیں کہ آپ **while** کے ساتھ **int** یا **char** ویری ایبل ہی استعمال کریں۔ ہم **while** کے ساتھ **Float** بھی استعمال کر سکتے ہیں۔ جیسے

```
float i=10.0;
while(i<=10.5)
{
printf ("%f",i);
i=i+0.1;
}
```

7- ہم **loop** کے **counter** کو بریکٹس میں بھی استعمال کر سکتے ہیں۔ جیسے

```
while(++i<=10);
```

8- ضروری نہیں کہ ہم کوئی ویری ایبل ہی استعمال کریں۔ہم Loop کو کسی decision کی مدد سے بھی ختم کر سکتے ہیں (جس پر ہم بعد میں بات کریں گے)۔ایسی صورت میں ہم یوں بھی لکھ سکتے ہیں۔

```
while(1)
do-this;
```

1 سے مراد true ہے لہٰذا Loop چلتا رہے گا۔

امید ہے آپ while کو سمجھ گئے ہوں گے۔ آیئے do...while پر بات ہو جائے۔

do...while

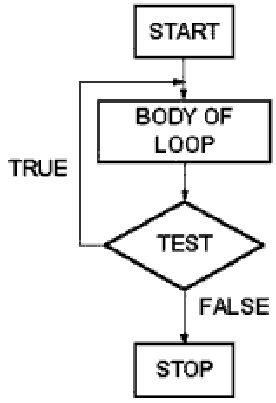
syntax:

```
do
{
do-this;
do-this;
} while (condition);
```

while loop اور do...while میں بنیادی فرق یہ ہے کہ while میں کنڈیشن پہلے چیک ہوتی ہے اور do..while میں بعد میں۔

کچھ حالات میں ہم چاہتے ہیں کہ loop کسی بھی صورت میں کم ازکم ایک بار ضرور چلے تو ہم while کے بدلے do....while کو استعمال کرتے ہیں۔

do...while کا Loop بھی True پر چلے گا اور FALSE پر ختم ہوگا۔فلو چارٹ ملاحظہ فرمایئے:



آیئے پروگرام سے سمجھتے ہیں:

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
int n=0;
do
{
printf("enter no:");
scanf("%d",&n);
} while(n!=0);
}
```

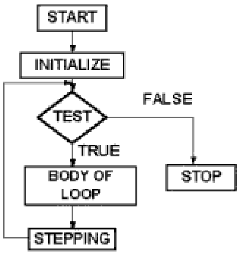
یاد رکھیں، do...while میں while کے بعد آپ semicolon لازمی دیں گے تاکہ کمپائلر سمجھ سکے کہ یہ do کی ending ہے نہ کہ کسی نئے while کی شروعات۔
لیجئے جناب، do...while بھی ختم ہوا۔ آیئے اب آخری looping structure کو بھی ڈسکس کر لیتے ہیں جو مقررہ تعداد تک چلتا ہے۔

for loop

syntax:

```
for(initialization; testing; stepping)
do-this;
```

اَب اِس لُوپ کا فلو چارٹ بھی دیکھتے چلیے:



for loop کے اندر 3 حصے ہوتے ہیں۔
initialization: اس حصے میں ہم Loop counter کی ابتدائی قیمت مقرر کرتے ہیں۔
Testing: اس حصے میں ہم condition بتاتے ہیں جس کی مدد سے Loop یہ چیک کرتا ہے کہ کاؤنٹر کی مقررہ قیمت حاصل ہو چکی ہے یا نہیں۔
Stepping: اس سے مراد ہے testing ویری ایبل کی ویلیو کا تبدیل ہونا، خواہ زیادہ ہو یا کم۔
آیئے for loop کو پروگرام سے سمجھ لیتے ہیں۔

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
int i;
for(i=0; i<=3; i++)
    printf("ABC\n");
}
```

اس پروگرام کا output کچھ یوں آئے گا

ABC
ABC
ABC
ABC

جی ہاں! Loop چار مرتبہ چلے گا، 0 سے 3 تک۔
i ویری ایبل کو ہم نے کاؤنٹر کے طور پر استعمال کیا ہے اور اس کی ابتدائی ویلیو صفر دی ہے۔
Test کنڈیشن کے مطابق Loop تب تک چلے گا جب تک کہ کاؤنٹر کی ویلیو 3 سے کم یا اس کے برابر ہوگی۔ ایک اور مثال دیکھیے۔ ہم 5 سے 1 تک نمبرز کو نزولی ترتیب (Descending order) میں پرنٹ کروانا چاہ رہے ہیں۔

```
for(i=5; i>=1; i--)
    printf("%d",i);
```

for loop، Syntax کے اعتبار سے بہت زیادہ آسان ہے۔ ہم اسے مختلف حالتوں میں مختلف طریقوں سے استعمال کر سکتے ہیں۔ مثلاً:
1- اگر آپ نے ویری ایبل پہلے سے نہ بنایا ہو تو for کے بریکٹس میں ہی Declare کر سکتے ہیں۔

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

2- ایک سے زائد Initialization بیک وقت ممکن ہے۔ ہم انہیں کوما (,) آپریٹر سے الگ کریں گے۔

```
for(i=0, j=1, k=2; i<=3; i++)
```

3- ایک سے زائد stepping بھی ممکن ہے۔

```
for (i=0,j=1; i<=3; i++, j=j+2)
```

4- اگر ہم ویری ایبل کو Decleration کے وقت ہی کوئی ویلیو assign کر چکے ہیں تو initialization بے مقصد ہو جاتی ہے۔ ایسی صورت میں ہم initialization پارٹ کو یکسر ختم کر دیتے ہیں۔ جیسے

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

5- ضروری نہیں کہ Stepping بریکٹس کے اندر ہی کی جائے۔ ہم اسے loop body میں بھی کر سکتے ہیں۔ جیسے

```
for(; i<=3;)
{
____
------
____
i++;
}
```

6- stepping اور testing پارٹ کو combine بھی کیا جا سکتا ہے۔ جیسے

```
for(; i++<=3;)
```

7- اگر آپ loop کو بعد میں (Loop body) کسی decision کی مدد سے ختم کرنا چاہ رہے ہوں تو for کے بریکٹس میں کچھ نہیں آئے گا۔

```
for (;;)
```

یہ syntax بھی allow ہے۔ مگر اس صورت میں infinite loop چلے گا۔

آیئے ایک پروگرام بناتے ہیں جو یوزر سے ایک نمبر ان پٹ لے اور دیئے جانے والے نمبر کا table ڈسپلے کردے۔

```
#include <conio.h>
#include <stdio.h>
void main(void)
{
int n;
printf("Enter any no:");
scanf("%d",&n);
for(int i=1; i<=10; i++)
        printf("%d",n*i);
getch();
}
```

چلئے اب آپ بتایئے کہ مندرجہ ذیل پروگرام کا output کیا آئے گا۔

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
for(int i=0; i<=5; i++);
        printf("%d",  i);
getch();
}
```

اس پروگرام کا output آئے گا 6

صرف 6، جی ہاں! کیونکہ Loop کے فوراً بعد ہم نے ; سیمی کولن لگا دیا ہے، جس کی وجہ سے یہ ایک null loop بن گیا ہے loop کچھ نہیں کرے گا سوائے counter کو increase کرنے کے۔ جب کاؤنٹر کی ویلیو 6 ہو جائے گی تو loop ختم ہو جائے گا اور کنٹرول loop کے بعد والی stetement پر ٹرانسفر ہو جائے گا۔ نتیجتاً صرف 6 پرنٹ ہوگا۔

BREAK STATEMENT

یہ statement جیسے ہی Run ہوگی ہمیں اسٹرکچر سے باہر نکال دے گی۔ آپ اسے پہلے ہی switch میں دیکھ چکے ہیں۔ ایک مثال سے اس کا استعمال سمجھ لیتے ہیں۔
آپ نے while کو سمجھتے ہوئے ایک پروگرام بنایا تھا کہ پروگرام یوزر سے کیریکٹرز ان پٹ لے جب تک کہ یوزر enter پریس نہ کردے۔ اب آپ اس پروگرام کو دوبارہ لکھئے، while کے بجائے for کے ذریعے۔

```
for(;;)
if(getche()== `\r`) break;
```

CONTINUE STATEMENT

سب سے پہلے یہ بات ذہن نشین کو لیجئے کہ break کو ہم Loop اور switch دونوں کے ساتھ استعمال کر سکتے ہیں جبکہ continue کو صرف loops کے ساتھ ہی استعمال کیا جائے گا۔ continue جیسے ہی Run ہوگی یہ فوراً کنٹرول کو Loop کے Header پر ٹرانسفر کر دے گی۔ مندرجہ ذیل پروگرام سے سمجھیں:

```
#include <conio.h>
#include <stdio.h>
void main(void)
{
int i, sum=0;
for(i=1; i<=5; i++)
{
if(i==2 || i==3)
continue;
sum+=i;
}
printf("%d",sum);
}
```

اس پروگرام کا output آئے گا 10۔ کیونکہ جیسے ہی counter کی ویلیو 2 یا 3 ہوگی continue چل جائے گی اور کنٹرول مزید کوئی statement انجام دیے بغیر loop header پر ٹرانسفر ہو جائے گا۔ چلئے اب آپ مندرجہ ذیل پروگرام کا output بتایئے۔

```
for(i=1; i<=2; i++)
   for(j=1; j<=2; j++)
  {
        if(i==j) continue;
        printf("%d\n",j);
  }
```

output

2
1

ایک اور پروگرام دیکھئے۔

```
for(i=1; i<=2; i++)
  for(j=1; j<=2; j++)
  {
        if(i==j) break;
        printf("%d\n",j);
  }
```

output

1

امید ہے آپ تینوں Loop اور Break اور continue کا استعمال سمجھ چکے ہوں گے۔

ایک بات اور ذہن نشین کر لیں۔ وہ یہ کہ C اور C++ میں بنیادی فرق یہ ہے کہ ہم C++ میں آبجیکٹ اورینٹڈ پروگرامنگ (OOP) کا استعمال کرتے ہیں مگر جب تک کہ آپ اسٹرکچرڈ پروگرامنگ سے واقفیت حاصل نہ کریں گے OOP آپ کے لئے سعی لاحاصل ہے اسی لئے ہم فی الحال اسٹرکچرڈ پروگرامنگ کو ہی ڈسکس کر رہے ہیں۔

باب نمبر 8
فنکشنز (Functions)

یوں تو ++C میں فنکشن کو بہت سے طریقوں سے استعمال کیا جاتا ہے۔ کچھ ہم ابھی زیر بحث لائیں گے اور کچھ پر آبجیکٹ اورینٹڈ پروگرامنگ (OOP) کے حصے کے ساتھ بحث کریں گے۔ فنکشن سے متعلق جن موضوعات پر ہم اس وقت بات کریں گے، وہ یہ ہیں:

فنکشن سے کیا مراد ہے؟	فنکشن کی ضرورت
فنکشن کی افادیت	فنکشن کی قسمیں
فنکشن لکھنے کا طریقہ کار	فنکشن کے مختلف حصے
	فنکشن کے استعمالات

فنکشن کی منطق (Logic)

روزمرہ زندگی میں ہم مختلف کام مختلف لوگوں سے کرواتے ہیں۔ انسان ایک بہترین دماغ اور بے شمار صلاحیتوں کے باوجود بہت سے کاموں میں دوسروں کا محتاج ہے۔ تمام تر کام خود سے نہیں کرسکتا۔ مثلاً کار کو ٹھیک کروانے کے لئے ہم مکینک بلاتے ہیں، باغ کی دیکھ بھال کے لئے مالی ضروری ہے۔ الغرض آپ کسی بھی طرف دیکھ لیں، آپ کو تمام انسانوں کا کسی نہ کسی طور پر باہم انحصار نظر آئے گا۔ ایک آفس کی مثال لے لیں چپراسی، کلرک، اکاؤنٹینٹ، آفیسر، ہر کام کے لئے لوگ مخصوص ہیں۔ ایک آدمی جو کہ سربراہ یا Boss ہوتا ہے، تمام آدمیوں کو ان کے مخصوص کام دیتا اور ان سے رپورٹ لیتا ہے۔ آپ خود سوچیے کہ اگر ایک ہی آدمی تمام کام کرتا تو رزلٹ میں کتنی تاخیر ہو جاتی اور کیا ایک ہی آدمی ہر قسم کے کام کے لئے موزوں ہوتا؟ لہٰذا ہم نے روزمرہ کے مختلف کاموں کو مختلف لوگوں میں تقسیم کر دیا اور ایک آدمی کو ان سب کا نگران بنا دیا۔ بالکل اسی کلیہ کو سامنے رکھتے ہوئے ہم نے پروگرامنگ لینگویجز میں فنکشن کو ترتیب دیا۔

اب تک آپ نے جو بھی پروگرام بنائے ان میں صرف ایک ہی فنکشن ہوتا تھا، جس کا نام ()main تھا۔

اب ہم ایک ہی پروگرام میں مختلف کاموں کے لئے مختلف فنکشن بنا دیں گے اور ان سب کا نگران ہوگا ()main فنکشن۔ ()main سے ہی تمام فنکشن call ہوں گے اور ()main کو ہی تمام فنکشن رپورٹ Return کریں گے۔ یہ تو تھی فنکشن کی منطق، اب دیکھتے ہیں کہ فنکشن کہتے کسے ہیں۔

فنکشن کا مطلب

سب سے پہلے تو فنکشن کو فنکشن ہی کیوں کہتے ہیں۔

روزمرہ زندگی میں تو شادی بیاہ، گانے بجانے، تقریر یا مقابلے کے انعقاد کو ہم فنکشن کہتے ہیں۔

تو ++C کی Statements کا مندرجہ بالا عوامل سے کیا تعلق کہ ہم دونوں کو ایک ہی نام سے موسوم کرتے ہیں۔

فنکشن کے معنی ہوتے ہیں کسی کام کو کرنے کا مناسب طریقہ کار

"PROPER ACTIVITY TO DO SOMETHING"

روزمرہ زندگی میں فنکشن ترتیب وار مراحل کا نام ہے جن کے بعد کوئی کام پایہ تکمیل تک پہنچے۔ یہی مشابہت ہے روزمرہ زندگی اور ++C کے فنکشن میں۔

++C میں بھی جیسے ہی کوئی فنکشن Call ہوتا ہے کچھ Statements چلتی ہیں اور بالآخر ہمیں کوئی Output موصول ہو جاتا ہے۔

کمپیوٹر سائنس میں فنکشن سے مراد ایک sub-program ،Unit یا کمانڈز کا وہ مجموعہ ہوتا ہے جس کی طرف ہم کچھ اِن پُٹ کریں۔ وہ دیئے جانے والے اِن پُٹ پر عمل کرے اور اس عمل کے بعد جو نتیجہ نکلے وہ ہمیں output کے طور پر واپس کر دے۔

اب ++C میں یہ بات ضروری نہیں کہ ہم فنکشن کی طرف لازمی کوئی Input پاس کریں یا وہ لازماً ہمیں کوئی آؤٹ پُٹ Return کرے۔

لہٰذا ++C میں فنکشن کی تعریف کچھ یوں ہوگی:

"کمانڈز کا وہ مجموعہ جو کوئی کام سرانجام دے، جبکہ اسے Call کیا جائے۔"

(Grouping of statements that carry out some task at calling level.)

مثال کے طور پر ایک دکان ہے جس میں ایک شخص کو ملازم رکھا گیا ہے۔ اس کا کام یہ ہے کہ جب بھی کوئی بل اس کے پاس آئے وہ اسے چیک کرے اور رقم لے کر دستخط کر دے۔

اب اس شخص (Function) کا یہ کام اسے بتا دیا گیا ہے۔ اب جیسے ہی کوئی بل اس کے پاس آئے گا تو اسے کچھ بتانے کی ضرورت نہیں۔ وہ اپنا کام خود انجام دے گا۔

بالکل اسی طرح ہم ++C میں بار بار دہرائے جانے والے کاموں کے لئے الگ الگ فنکشن بنا لیتے ہیں تاکہ بوقتِ ضرورت صرف انہیں call کیا جائے اور کام ہو جائے۔

فنکشن کی ضرورت

فنکشن کی مدد سے ہم پروگرام کو منطقی طور پر مختلف چھوٹے چھوٹے حصوں میں بانٹ دیتے ہیں جس کی وجہ سے پروگرام کو لکھنا، پڑھنا اور سمجھنا بہت زیادہ آسان ہو جاتا ہے۔

مثال کے طور پر مندرجہ ذیل **equation** کو دیکھیں:

A+B-C/D%F

اب ہم اس کو مختلف حصوں میں توڑ دیتے ہیں:

(((A+B)-C)/D)%F)

آپ دیکھئے کہ اب پرابلم کو سمجھنا اور حل کرنا کتنا آسان ہو گیا ہے۔
پروگرام کو مختلف فنکشن میں تقسیم کرنے سے پروگرام کی ٹیسٹنگ، **debugging** اور **execution** آسان ہو جاتی ہے۔

فنکشن کی افادیت

مثال کے طور پر آپ کو ایک پروگرام میں **FACTORIAL** معلوم کرنا اور پروگرام میں مختلف جگہوں پر متعدد بار مختلف اعداد کے ساتھ یہی عمل دہرانا ہے۔ اب اگر آپ ہر مرتبہ **factorial** معلوم کرنے کا پروگرام لکھتے رہے تو پروگرام کا سائز بھی بڑھ جائے گا اور بار بار ایک ہی کوڈ لکھنے کی کوفت الگ۔ لہٰذا ہم ایک فنکشن بنالیں گے **factorial** معلوم کرنے کا، اور پھر جہاں جہاں ضرورت پڑے صرف اسے **call** کرلیں گے۔ نہ پروگرام کا سائز بڑھا اور نہ ہی کوڈ بار بار لکھنے کی زحمت ہوئی۔
فنکشن کی مدد سے ہم ایک ہی کوڈ بار بار لکھنے سے بچ جاتے ہیں۔ یہاں ایک بات اور ذہن نشین کر لیجئے کہ اگر آپ کو کوئی کوڈ صرف ایک ہی بار استعمال کرنا ہو تو بھی اس کا الگ فنکشن ضرور لکھیں کیونکہ پروگرام کو منطقی اعتبار سے تقسیم کرنا ہر لحاظ سے بہتر ہے۔
فنکشن کی مدد سے ہم پروگرام میں غلطیوں کو آسانی سے ڈھونڈ سکتے اور انہیں دور کر سکتے ہیں۔ مثلاً آپ نے ہزاروں لائنوں پر محیط ایک پروجیکٹ بنایا۔ تمام تر کام بخوبی انجام پا رہے ہیں مگر تاریخ درست طور پر ڈسپلے نہیں ہو رہی۔ اب آپ کو ہزاروں لائنوں میں سر کھپانے کی قطعاً ضرورت نہیں۔ آپ صرف **Date** والے فنکشن کو درست کرلیں گے۔
یہ ضروری نہیں کہ آپ فنکشن کو صرف ایک ہی پروگرام میں **Call** کرسکیں۔ ایک ہی فنکشن کو ہم مختلف پروگراموں میں بھی **call** کرسکتے ہیں۔

فنکشن کی قسمیں

بنیادی طور پر فنکشن کی دو قسمیں ہوتی ہیں:

1- Built-in or library function

2-user defined functions(udf)

1۔ بلٹ ان/لائبریری فنکشن

C++ میں ہمارے پاس تقریباً **3500** (ساڑھے 3 ہزار) فنکشن موجود ہیں جو مختلف **header files** کے ذریعے ہم **call** کرسکتے ہیں۔ ایسے فنکشن بلٹ ان یا **standard function** کہلاتے ہیں۔ ان فنکشن کو ہم صرف کال کرتے ہیں ان کو **define** کرنے کی کوئی ضرورت نہیں ہوتی۔ یہ فنکشن پہلے ہی سے **defined** ہوتے ہیں۔ جب کوئی فنکشن کال کیا جائے تو **C++** کمپائیلر لائبریری فائلوں سے پڑھ لیتا ہے کہ کیا کرنا ہے اور فنکشن کا ایڈریس وہ **Header file** سے حاصل کرتا ہے۔
Built-in فنکشن کو مندرجہ ذیل **17** کیٹیگریز میں تقسیم کیا گیا ہے:

1- Arithmatic functions
2- Data conversion functions
3- Character classification functions
4- String manipulation functions
5- Searching functions
6- Sorting functions
7- I/O functions
8- File handling functions
9- Directory control functions
10- Buffer Manipulation functions
11- Disk I/O functions
12- Memory Allocation functions
13- Process control functions
14- Graphic functions
15- Time Related functions
16- DOS interface functions
17-Miscellaneous functions

2۔ یوزر ڈیفائنڈ فنکشن

وہ فنکشن جن کا کوڈ یوزر اپنے من پسند کام کو انجام دینے کے لئے خود لکھتا ہے انہیں ہم یوزر ڈیفائنڈ فنکشن یا **udf** کہتے ہیں۔

فنکشن کے مختلف حصے

آپ کوئی بھی فنکشن لکھیں، اس میں مندرجہ ذیل 3 حصے ضرور ہوں گے:

1- Function Prototype part

2- Function Calling part

3- Function Definition part

فنکشن لکھنے کا طریقہ کار (syntax)

آیئے مذکورہ بالا تینوں Parts کو سمجھ لیتے ہیں۔

فنکشن پروٹوٹائپ

proto کا مطلب ہوتا ہے First یا پہلا۔
type سے مراد ماڈل یا نمونہ ہے۔

جس طرح کوئی بھی بلڈنگ بنائی ہو تو پہلے اس کا نقشہ اور ماڈل بنتا ہے۔ آپ شوکیس میں رکھے ہوئے ماڈل سے بلڈنگ کا اندازہ بخوبی کر سکتے ہیں کیونکہ ماڈل، بلڈنگ میں موجود ہر ہر چیز کی بھرپور عکاسی کر رہا ہوتا ہے۔ اس میں لفٹ، کار پارکنگ، شاپنگ مال غرض ہر چیز ہمیں نظر آ رہی ہوتی ہے۔ بالکل اسی طرح کوئی بھی فنکشن لکھنے سے پہلے ہم اس کا پروٹوٹائپ ترتیب دیتے ہیں۔ یہ پروٹوٹائپ فنکشن کو استعمال کرنے اور سمجھنے میں کمپائلر کی مدد کرتا ہے۔

Prototype کی تعریف کچھ یوں ہوگی: ''پروٹوٹائپ فنکشن کی توثیق (Validation) کے لئے استعمال ہوتا ہے''۔

مثال کے طور پر ہم نے ایک فنکشن لکھا جس کا نام ABC رکھ دیا اب ABC نام کا کوئی لائبریری فنکشن تو ہے نہیں۔ تو کمپائلر اس پر Error کیوں نہیں دے رہا؟
Prototype کی وجہ سے۔ یعنی ہم نے ABC فنکشن کا پروٹوٹائپ بنا کر اسے پروگرام میں شامل کر دیا ہے۔
آپ پروگرام میں clrscr() لکھیں اور conio.h کو include نہ کروائیں تو کمپائلر error دے گا۔

"clrscr() should have a prototype"

Prototype syntax

function_type function_name

(Parameter list);

فنکشن ٹائپ (function type)

یہاں ہم وہ ڈیٹا ٹائپ دیتے ہیں جس ڈیٹا ٹائپ کی ویلیو فنکشن ریٹرن کرے گا۔ مثال کے طور پر ایک فنکشن ہے جو جمع کرتا ہے تو ہم عموماً اس کی ٹائپ int ہی دیں گے۔
فنکشن ٹائپ کو ہم Return type بھی کہتے ہیں۔ اگر ہم نے ایسا فنکشن بنایا ہے جو صرف ایک مخصوص کام سرانجام دیتا ہے اور ریٹرن کچھ بھی نہ کرے تو اس کی Return ٹائپ void ہوگی۔

فنکشن کا نام

فنکشن کا نام کوئی بھی ہو سکتا ہے جو ویری ایبل کے اصولوں کے مطابق ہو۔ یعنی کہ پہلا کریکٹر نیومیرک نہ ہو کیریکٹر ہو۔ کوئی اسپیشل سمبل، نام کے درمیان نہ ہو سوائے انڈر اسکور (_) کے۔ فنکشن کے نام کے فوراً بعد آپ Paranthesis() لکھ دیں گے۔ دونوں کے درمیان کوئی Space نہ ہو۔
فنکشن کا نام رکھتے ہوئے کوشش کریں کہ نام فنکشن کے کام کی مناسبت سے ہو۔ مثال کے طور پر جمع کے فنکشن کا نام آپ sum یا add رکھ دیں۔
ایک بات اور ذہن نشین کر لیجئے کہ فنکشن کا نام کسی بلٹ اِن فنکشن کا نام نہ ہو۔ آپ printf اور scanf نام کے فنکشن نہ بنائیں، یا کسی بھی Built-in فنکشن کے نام سے نیا فنکشن نہ بنائیں۔

پیرامیٹر لسٹ (Parameter List)

پیرامیٹر لسٹ سے مراد ان پٹ کی وہ فہرست ہے جو ہم فنکشن کی طرف پاس کریں گے۔
اب جتنے ان پٹ ہم فنکشن کی طرف پاس کریں گے، اتنی ہی Parameters کی تعداد ہوگی۔ یہاں ہم پر Parameters کی Data type بتا دیتے ہیں تاکہ کمپائلر کو پتا چل سکے کہ پاس ہونے والے Parameters کی تعداد اور ان کی ڈیٹا ٹائپ کیا ہے۔
اگر ہم فنکشن کی طرف کوئی ان پٹ پاس نہ کرنا چاہیں تو اس صورت میں ہم پیرامیٹر لسٹ میں void لکھ دیں گے۔ void کا مطلب ہوتا ہے ''کچھ نہیں۔''
آیئے Parameters کو مزید سمجھ لیں۔
Parameters کسے کہتے ہیں: وہ یونٹس جن کی مدد سے ہم ڈیٹا کو پروگرام کے ایک حصے سے دوسرے حصے کی طرف منتقل کریں، انہیں ہم Parameter کہتے ہیں۔

Parameters کی قسمیں

بنیادی طور پر Parameters کی 3 قسمیں ہوتی ہیں۔

1- Input parameters

2- Output parameters

3- Formal parameters

input Parameters

یعنی وہ پیرامیٹرز جن کی مدد سے ڈیٹا main()یا Calling function سے کال کیے جانے والے فنکشن کی طرف پاس ہو۔

Output parameters

یعنی جن کی مدد سے ڈیٹا، کال کیے جانے والے فنکشن سے کال کرنے والے فنکشن یا main() کی طرف Return ہو۔

Formal parameters

فنکشن کے اپنے Variables جہاں وہ ملنے والے ان پٹ کی values کو بعد میں استعمال کیلئے store کر سکے، انہیں ہم Formal Arguments کہتے ہیں۔

یہاں Parameters یا Arguments دونوں الفاظ ایک ہی معنی میں استعمال ہوتے ہیں۔ یقیناً اب Prototype کا syntax سمجھ میں آ گیا ہوگا۔

آیئے دیکھتے ہیں کہ Prototype، کمپائیلر اور ہمیں فنکشن سے متعلق کیا کیا بتاتا ہے۔ پروٹوٹائپ، کمپائیلر کو بتاتا ہے:

1- فنکشن ٹائپ　　　2- فنکشن کا نام

3- پیرامیٹرز کی تعداد　　　4- پیرامیٹرز کی ڈیٹا ٹائپ

5- پیرامیٹرز کی ترتیب

اگر ہم کال کرتے ہوئے مندرجہ بالا میں سے کسی بھی چیز کا خیال نہیں رکھیں گے تو کمپائیلر error دے گا۔

مثلاً: فنکشن کا نام غلط لکھ دیں، پیرامیٹرز کم یا زیادہ دے دیں، پیرامیٹرز کی ڈیٹا ٹائپ کا خیال نہ رکھیں یا ان کی ترتیب بدل دیں۔ مثال کے طور پر پہلا پیرامیٹر کیریکٹر اور دوسرا integer کا دے دیں جبکہ اصل ترتیب اس کے برعکس ہو۔ اس بات کو آپ مزید تب سمجھیں گے جب ہم پروگرام بنائیں گے۔

Function calling

فنکشن کا دوسرا پارٹ calling ہے۔ اس کا syntax بہت زیادہ آسان ہے۔ پروگرام میں کسی بھی جگہ صرف فنکشن کا نام لکھیں اور فنکشن کال ہو گیا۔

فنکشن کال کرتے ہوئے Parenthesis() کے بعد سیمی کولن (;) لگانا نہ بھولیں۔

Function Definition

اس پارٹ میں ہم فنکشن کو ڈیفائن کرتے ہیں یا اس کا مکمل کوڈ لکھتے ہیں کہ جب فنکشن کال ہوگا تو کیا کام انجام دے گا۔

Functions کی مدد سے ہم پروگرام کو مختلف حصوں میں تقسیم کرتے ہیں اور اسی تقسیم کا نام Structured programming یا Modularity ہے۔

ہاں تو جناب! تھیوری بہت ہو گئی۔ آیئے اب پروگرام بنالیں۔ آپ فنکشن کی مدد سے ایک پروگرام بنانا چاہ رہے ہیں کہ جب فنکشن کو call کریں تو وہ (*)Asterik کی ایک لائن پرنٹ کر دے جسے آپ پروگرام میں کال کریں۔ اب اس task میں نہ تو کوئی ان پٹ پاس کرنے کی ضرورت ہے اور نہ ہی کوئی ویلیو ریٹرن ہوگی۔

```
#include <stdio.h>
#include <conio.h>
void starline(void);          ——— 1
void main(void)
{
  starline();                 ——— 2
}
void starline(void)
{
printf("***********");        ——— 3
}
```

1-Prototype پروٹوٹائپ　　　2- Calling　　　3- Definition

سب سے پہلے ہم نے پروٹوٹائپ لکھی۔ starline ہمارے بنائے ہوئے فنکشن کا نام ہے۔ آپ xyz,ABC کچھ بھی رکھ سکتے ہیں۔

فنکشن کال کرتے ہوئے صرف اس کا نام لکھ دیا۔ اور آخر میں فنکشن کو ڈیفائن کر دیا کہ وہ کیا کرے۔

جب یہ پروگرام چلے گا تو کمپائیلر ہمیشہ سب سے پہلے main() کو ہی پڑھے گا۔ main() فنکشن سے Starline کال ہوگا۔ پروگرام کا کنٹرول starline کی طرف پاس ہو جائے گا۔ printf() کی مدد سے Asterik کی لائن پرنٹ ہو گی اور control واپس main() کی طرف منتقل ہو جائے گا؛ اور پروگرام ختم ہو جائے گا کیونکہ starline کے بعد کوئی بھی کمانڈ نہیں۔ آپ پروگرام کا output دیکھنے کیلئے starline کے بعد getch() کی کمانڈ لکھ دیجئے۔ امید ہے پروگرام سمجھ میں آ گیا ہوگا۔

ایک اور پروگرام بناتے ہیں۔ ہم چاہ رہے ہیں کہ دو ویلیوز فنکشن کی طرف پاس کریں۔ وہ دونوں کو جمع کر کے ہمیں جواب ریٹرن کر دے۔ اب یہاں فنکشن ٹائپ int ہوگی، پیرامیٹرز کی تعداد دو ہوگی اور دونوں کی ٹائپ بھی int ہوگی۔ پروگرام دیکھئے:

```
#include <stdio.h>
#include <conio.h>
int sum(int , int);          ———— 1
void main(void)
{
clrscr();
int r;
r=sum(5,10);                 ———— 2
printf("%d",r);
getch();
}
int sum(int a, int b)        ]
{                            ]———— 3
return (a+b);                ]
}
```

ایک سے زائد Parameters کو ہم کوما آپریٹر (,) کی مدد سے الگ کرتے ہیں۔

فنکشن کی طرف دو ویلیوز 5 اور 10 ہم نے پاس کیں۔ ان ویلیوز کو فنکشن نے بالترتیب a اور b ویری ایبل میں محفوظ کر دیا۔

آپ ویری ایبل کے نام Prototype میں بھی دے سکتے ہیں۔ مثلاً

```
int sum(int a,int b);
```

اگر ہم Return type کچھ بھی نہ دیں تو By default کمپائلر اسے int ہی تصور کرے گا۔

مندرجہ ذیل دونوں Prototype یکساں ہیں۔

```
int sum(int,int);
sum (int,int);
```

Return statement

Return statement کنٹرول کو واپس وہیں ٹرانسفر کر دیتی ہے جہاں سے فنکشن کال کیا گیا تھا، اور ہم نے جو ویلیو دی ہوتی ہے اسے بھی ساتھ ہی منتقل کر دیتی ہے۔

اب اس پروگرام میں a+b کو جب Solve کیا جائے گا تو جواب آئے 15 جو ہم نے Return کروایا ہے۔

15 کو ہم نے r میں محفوظ کیا اور ڈسپلے کروا دیا۔

اس پروگرام میں ان پٹ پیرامیٹرز 5 اور 10 ہیں؛ a اور b، Formal parameters ہیں؛ اور output parameter ہے 15۔

فنکشن کو اگر اس طرح کال کیا جائے کہ کال کرتے ہوئے ہم constant ویلیوز پاس کریں تو اس طریقہ کار کو call by constant کہتے ہیں۔

اور اگر call کرتے ہوئے Variables استعمال کریں تو یہ طریقہ کار Call by Variable کہلائے گا۔ جیسے:

```
int i=5, j=10;
r=sum(i , j);
```

دونوں صورتوں میں جواب یکساں آئے گا۔ i اور j کی ویلیوز بالترتیب a اور b میں Copy ہو جائیں گی۔ چلئے ایک اور پروگرام بناتے ہیں:

ہم چاہ رہے ہیں کہ فنکشن کی طرف ایک کیریکٹر اور ایک نمبر پاس کریں۔ دیے جانے والے کیریکٹر کو فنکشن، دی جانے والی تعداد میں پرنٹ کر دے۔ اس پروگرام کو آپ خود دیکھئے:

```
#include <conio.h>
#include <stdio.h>
void repchar(int, char);     ———— 1
void main(void)
{
repchar(10, '+');            ]
repchar(25, '*');            ]———— 2
repchar(35, '$');            ]
getch();                     ]
}
void repchar(int j,char ch)  ]
{                            ]
for(int i=0; i<j; i++)       ]———— 3
        printf("%c",ch);     ]
printf ("\n");               ]
}
```

اس پروگرام میں ہم نے ایک ہی فنکشن کو مختلف ویلیوز کے ساتھ 3 بار کال کیا ہے۔

باب نمبر 9
ایڈوانسڈ فیچرز

ایگریگیٹ فنکشن(Aggregate Function)

ایسا فنکشن جو آپ کو پاس کی جانے والی بہت سی ویلیوز میں سے ایک ویلیو ریٹرن کرے۔ مثال کے طور پر آپ نے فنکشن کی طرف 3 نیومیرک ویلیوز پاس کیں۔ اب وہ تینوں کو جمع کر کے آپ کو ایک ویلیو ریٹرن کر دیتا ہے تو وہ ایگریگیٹ کہلائے گا۔

فنکشن اوورلوڈنگ(Function Overloading)

فنکشن کا مطلب تو آپ کو پتا ہی ہے کہ کمانڈز کا وہ مجموعہ جو کوئی منتخب کام سرانجام دے جب اسے call کیا جائے۔ آیئے overloading کو سمجھ لیں۔ روزمرہ زندگی میں کسی بھی چیز کو ضرورت سے زیادہ زیر بار کرنا overloading کہلاتا ہے۔ بالکل اسی طرح اگر ہم C++ میں ایک فنکشن کو ایک سے زائد کاموں کے لئے استعمال کریں تو وہ Over Loaded فنکشن کہلائے گا۔

Function overloading سے مراد ہے ایک ہی فنکشن کو مختلف کاموں کے لئے استعمال کرنا۔ Function اوورلوڈنگ ایک منطقی ذریعہ ہے ایک ہی فنکشن کو مختلف ڈیٹا ٹائپ، Arguments یا مختلف syntax کے ساتھ call کرنے کا۔

مثال سے سمجھیں: آپ نے ایک فنکشن بنایا اور اس کی طرف پاس کئے جانے والے Argument کی ڈیٹا ٹائپ بتائی int۔ اب اگر آپ فنکشن کی طرف char یا float کا پیرامیٹر پاس کریں گے تو error آجائے گا۔

مگر printf() بھی تو ایک فنکشن ہے۔ اس کے ساتھ ہم چاہیں تو int پاس کریں، چاہیں تو کیریکٹر، کوئی error نہیں آتا۔ یہ Function overloading کی ہی وجہ سے ممکن ہوا ہے کہ ہم ایک فنکشن کو مختلف Syntax کے ساتھ، مختلف طریقوں سے Call کرسکتے ہیں۔

عملاً اگر ہم function overloading کو دیکھیں تو اس کا مطلب ہوگا ایک ہی پروگرام میں ایک ہی فنکشن کو ایک سے زائد بار ڈیفائن کرنا۔ امید ہے آپ کو Function overloading کا مطلب اور مقصد دونوں سمجھ میں آگئے ہوں گے۔ آیئے! اب دیکھتے ہیں کہ فنکشن اوورلوڈنگ ہوتی کس طرح ہے۔

Syntax

فنکشن اوورلوڈنگ کا syntax بالکل عام فنکشن کی طرح ہے۔ وہی بنیادی 3 حصے ہیں۔ بس اس میں ایک ہی نام کا فنکشن ایک سے زائد مرتبہ ڈیفائن ہوگا۔

اب یہاں ایک سوال ذہن میں اٹھتا ہے کہ جب فنکشن کا نام ہی ایک ہوگا تو کمپائیلر بہت سی Definitions میں سے کس کو call کرے گا؟

Function overloading میں آپ کو صرف ایک خیال رکھنا ہے۔ وہ یہ کہ ایک ہی نام کے مختلف فنکشن کی Parameter list ضرور مختلف ہو، تاکہ کمپائیلر ڈیٹا ٹائپ کے حوالے سے جان سکے کہ کون سا فنکشن Call کیا گیا ہے۔

Function type یا Return کی تبدیلی سے کوئی فرق نہیں پڑتا۔ لازمی ہے کہ تبدیلی پیرامیٹرز لسٹ میں ہو، خواہ تعداد کی ہو، ڈیٹا ٹائپ کی یا ترتیب کی۔ آیئے، مندرجہ بالا تمام وضاحتوں کو پروگرام کی مدد سے سمجھ لیتے ہیں۔ ہم ایک پروگرام بنانا چاہ رہے ہیں کہ ایک فنکشن کو 3 مختلف syntax کے ساتھ call کرسکیں۔

فنکشن کا نام repchar() ہو، جب بغیر کسی پیرامیٹر کے call ہو تو ایک منتخب کیریکٹر، منتخب تعداد میں پرنٹ ہوجائے۔ جب صرف کیریکٹر کے ساتھ Call کریں تو وہ کیریکٹر منتخب تعداد میں پرنٹ ہوجائے۔ اور جب کیریکٹر اور تعداد، دونوں کے ساتھ call کریں تو دیا جانے والا کیریکٹر، دی جانے والی تعداد میں پرنٹ ہوجائے۔

پروگرام دیکھئے:

```
#include <conio.h>
#include <stdio.h>
void repchar();
void repchar( char );
void repchar( char, int);
void main(void)
{
clrscr();
repchar('+');
repchar('$',25);
getch();
}
void repchar()
{
for(int i=0; i<=35; i++)
        printf("*");
printf("\n");
}
void repchar(char ch)
{
for(int i=0; i<=35; i++)
        printf("%c",ch);
printf("\n");
}
void repchar(char ch, int j)
{
for (int i=0; i<=j; i++)
        printf("%c",ch);
printf("\n");
}
```

اس پروگرام میں ہم نے 3 فنکشن بنائے جن کے کام مختلف ہیں مگر تینوں کا نام ایک ہی رکھا تا کہ استعمال میں آسانی ہو۔ اسی کو ہم فنکشن اوور لوڈنگ کہیں گے۔ جب **repchar()** کال ہو گا تو کمپائیلر **calling** اور **prototype** کو آپس میں **Match** کرے گا۔ پہلی والی **prototype**، **match** ہو جائے گی اور پہلی **definition** کال ہو جائے گی۔

ایک بات اور ذہن میں بٹھا لیجئے کہ فنکشن اوورلوڈنگ اس وقت کہلائے گی جب ایک ہی نام کے بہت سارے فنکشن ہوں گے۔ اگر نام الگ الگ ہوئے تو وہ کسی بھی صورت اوور لوڈنگ نہیں ہوگی۔ امید ہے کہ آپ کو فنکشن اوورلوڈنگ سمجھ میں آ گیا ہوگا۔ اب آپ جلدی سے بتایئے کہ مندرجہ ذیل **prototype** درست ہے یا کمپائیلر **error** دے گا۔

```
int ABC();
void ABC();
```

کمپائیلر اس پر **error** دے گا، کیونکہ فنکشن اوورلوڈنگ کی لازمی شرط ہے کہ پیرامیٹر لسٹ **change** ہو۔ ریٹرن ٹائپ سے کوئی فرق نہیں پڑتا لہٰذا کمپائیلر کو پتا نہیں لگ سکتا کہ کون سا فنکشن کال کیا گیا ہے۔

ایک بات اور نوٹ کر لیجئے۔ وہ یہ کہ جب فنکشن کال ہوتا ہے تو کمپائیلر **calling** اور پروٹو ٹائپ کو آپس میں ملاتا ہے اور فنکشن کی **validation** کرتا ہے۔ جب پروگرام **Run** ہوتا ہے تو **Linker**، **calling,** اور **Definition part** کو **Match** کرتا ہے۔ اب آپ بتایئے کہ مندرجہ ذیل پروگرام میں کمپائیلر کون سا **error** دے گا۔

```
#include <conio.h>
#include <stdio.h>
void name();
void main(void)
{
name();
}
void name(char ch)
{
printf("%c",ch);
}
```

کمپائیلر کوئی **error** نہیں دے گا۔ جی ہاں! کمپائیلر **calling** اور پروٹو ٹائپ کو چیک کرتا ہے اور دونوں یکساں ہیں۔
ہاں! البتہ پروگرام **Run** نہیں ہوگا اور **Linker error** آ جائے گا جب آپ **Ctrl+F9** پریس کریں گے۔ مگر صرف **ALT+F9** پر کوئی **error** نہیں آئے گا۔

NAME MANGLING

MANGLE کہتے ہیں "صورت بگاڑ دینے، یا خراب کرنے" کو۔ فنکشن اوورلوڈنگ والے پروگرامز میں کمپائلر exe فائل بناتے ہوئے ایک ہی جیسے بہت سے فنکشن کو مختلف نام دے دیتا ہے، صرف آسانی سے سمجھنے کے لئے۔ اس صورت کو ہم Name Mangling کہتے ہیں۔

Name Mangling دراصل کمپائلر کا وہ Internal process ہے جس کی مدد سے وہ ایک ہی نام کے مختلف Functions میں تمیز کر سکتا ہے۔ فنکشن اوور لوڈنگ کو مزید سمجھنے کے لئے بات کرتے ہیں کمپائلر کے طریقہ کار پر کہ وہ کیسے overloaded فنکشن کو treat کرتا ہے۔ Calling اور Prototype کی Matching کا یہ طریقہ کار Argument Matching کہلاتا ہے۔

اس میں مندرجہ ذیل 3 صورتیں ہو سکتی ہیں:

(1) A MATCH

(2) NO MATCH

(3) Ambiguous MATCH

آیئے، تینوں صورتوں کو سمجھ لیتے ہیں۔

A MATCH

جب پاس کئے جانے والے پیرامیٹرز کی ڈیٹا ٹائپ بالکل وہی ہو جو ہم نے Prototype میں لکھی ہے۔ یا Formal اور Actual پیرامیٹرز بالکل ایک جیسے ہوں۔

NO MATCH

جب formal اور Actual پیرامیٹرز کی ٹائپ میں فرق ہو۔ مثال کے طور پر

```
void repchar(char); //Prototype
repchar(123); //Calling
```

ہم نے پروٹو ٹائپ میں کیریکٹر پیرامیٹر دیا اور call کرتے ہوئے integer پاس کر دیا۔ اس صورت میں کمپائلر error دے گا۔

Ambiguous Match

Ambiguous کا مطلب ہوتا ہے "مشکوک یا مبہم" ایسا کب ہوتا ہے؟ جب calling ایک سے زائد Prototype Match کر جائے۔ مثال کے طور پر دو Prototypes ہیں۔

```
void print(unsigned int);
void print(int);
```

اب اگر ہم ()print کو integer کے ساتھ call کریں گے تو کمپائلر Ambiguous Match کا error دے گا کیونکہ دونوں پروٹو ٹائپ Match کر رہی ہیں۔ کمپائلر اس صورتحال کو مندرجہ ذیل دو میں سے کوئی ایک طریقے سے حل کرتا ہے۔

(1) A Match through promotion

(2) A Match by standard conversion

آیئے، ان دونوں طریقوں کو سمجھ لیتے ہیں۔

A Match through Promotion

اس طریقہ کار میں اگر کوئی exact match نہیں ہوتا تو کمپائلر پہلے Step میں Actual پیرامیٹر کی Date type کو اپ گریڈ یا پروموٹ کر دیتا ہے۔ یہ طریقہ Promotion بھی کہلاتا ہے۔

Promotion کا طریقہ کار

1- ڈیٹا ٹائپ char,unsigned char, short پروموٹ ہو جائیں گے int میں۔ short یا تو int میں پروموٹ ہو گی یا unsigned int میں، اس کا انحصار آپ کی مشین پر ہے۔

2- float پروموٹ ہو گا Double میں۔

3- enumerated یا یوزر ڈیفائن ڈیٹا ٹائپ پروموٹ ہو گی int میں۔

A Match by standard conversion

اگر کوئی exact Match نہیں ہوتا اور Promotion کے بعد بھی کمپائلر فیصلہ نہیں کر پاتا تو یہ دوسرا Step عمل میں آتا ہے۔

1- Actual پیرامیٹرز کی نیومیرک ویلیو پروٹو ٹائپ کی کسی بھی نیومیرک ٹائپ سے Match ہو جائے گی خواہ وہ unsigned ہی کیوں نہ ہو۔

2- Enumerated, Match کریں گے نیومیرک کے ساتھ۔

3- زیرو ویلیو **Pointer** ٹائپ اور نیومیرک دونوں کو **Match** کرے گی۔

4- کسی بھی **type** کا **pointer**، **void pointer** سے **Match** ہو جائے گا۔

فنکشن اوورلوڈنگ تو ختم ہوئی۔ اب آیئے بات کرتے ہیں **Default Arguments** پر۔

ڈیفالٹ آرگیومنٹ

ڈیفالٹ آرگیومنٹ، پیرامیٹرز کی ہی ایک قسم ہے جس میں ہم پیرامیٹر لسٹ میں ڈیٹا ٹائپ دیتے ہوئے کوئی ڈیفالٹ ویلیو بھی بتا دیتے ہیں۔ اگر کوئی ویلیو پاس کی گئی تو ٹھیک ہے ورنہ پروٹو ٹائپ فنکشن کی طرف ڈیفالٹ ویلیوز ہی پاس کر دے گا۔ بالکل **Switch** کمانڈ کی طرح، اگر کوئی کیس **Match** ہو گیا تو ٹھیک ورنہ ڈیفالٹ ایگزیکیوٹ ہو جائے گا۔

Default Arguments درحقیقت **Function overloading** کی جدید شکل ہے۔ اس میں بھی ہم ایک ہی کمانڈ کو مختلف **Syntax** کے ساتھ لکھ سکتے ہیں یا ایک ہی فنکشن کو مختلف **syntax** کے ساتھ **call** کر سکتے ہیں۔

مگر اگر آپ سے **function overloading** کا پروگرام بنانے کو کہا جائے گا تو آپ وہی پروگرام بنائیں گے جو ہم ابھی بنا چکے ہیں کیونکہ **overloading** تو اسی وقت ہو گی جب ایک فنکشن کو ایک سے زائد بار ڈیفائن کیا جائے۔

چلیں پچھلے پروگرام کو **Default Arguments** کے ساتھ بناتے ہیں۔ آپ دیکھیں گے کہ پروگرام کوڈ کتنا مختصر ہو گیا مگر **output** بالکل وہی رہا۔

DEFAULT ARGUMENT PROGRAM

```
#include <conio.h>
#include <stdio.h>
void repchar(char='*', int=35);
void main (void)
{
repchar();
repchar('+');
repchar('$',25);
}
void repchar(char ch, int j)
{
for (int i=0; i<j; i++)
        printf("%c",  ch)
printf("\n");
}
```

جب ہم نے **repchar()** بغیر کسی پیرامیٹر کے کال کیا تو کمپائلر دونوں **formal** پیرامیٹرز کو ڈیفالٹ ویلیو پاس کر دے گا۔ جب صرف کیریکٹر پاس ہوا تو کمپائلر صرف **integer** کو ڈیفالٹ کی صورت میں پاس کر دے گا اور جب دونوں پیرامیٹرز پاس ہو گئے تو ڈیفالٹ ویلیو پاس نہیں ہو گی۔ امید ہے کہ **Default Arguments** سمجھ آ گئے ہوں گے نیز ایک بات یاد رکھیں کہ ڈیفالٹ آرگیومنٹ پروٹو ٹائپ کے مندرجہ ذیل دونوں **syntax** درست ہیں۔

```
(1) void repchar(char='*', int=35);

(2) void repchar(char ch='*', int j=35);
```

Inline functions

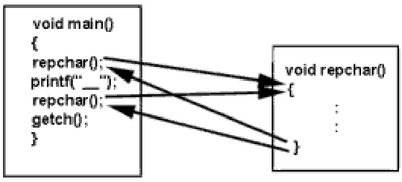
Inline فنکشن کو سمجھنے کیلئے ہمیں سب سے پہلے فنکشن کے کام کرنے کے طریقہ کار پر غور کرنا پڑے گا۔ مندرجہ ذیل پروگرام کو دیکھئے:

```
#include <conio.h>
#include <stdio.h>
void repchar();
void main(void)
{
clrscr();
repchar();
printf("ZEESHAN-UL-HASSAN");
repchar();
getch();
}
void repchar()
{
printf("-----------");
}
```

یہ پروگرام کس طرح **execute** ہو گا؟ کمپائلر سب سے پہلے **main()** کو پڑھے گا۔ اسکرین کلیئر ہو گی اور جیسے ہی **repchar()** کال ہو گا، پروگرام کا

کنٹرول ()repchar کے Definition Part کی طرف منتقل ہو جائے گا۔ایک لائن پرنٹ ہوگی اور فنکشن کے اختتام پر کنٹرول واپس main کی طرف Return ہوگا جہاں سے فنکشن call ہوا ہے۔اس کے بعد والی لائن پر ()printf کی مدد سے نام پرنٹ ہوگا پھر ()repchar کال ہوا اور پروگرام کا کنٹرول ایک مرتبہ پھر فنکشن کی طرف منتقل ہو گیا۔فنکشن کے اختتام پر کنٹرول ایک مرتبہ پھر main کی طرف ریٹرن ہوگا، پھر ()getch رن ہوگا۔

اب کمپائیلر کو ریٹرن ہوتے ہوئے کیسے پتا چلا کہ کون سی لائن پر ریٹرن ہونا ہے، پہلے فنکشن کے بعد والی لائن پر یا دوسرے فنکشن کے بعد والی لائن پر۔ یہ بات سمجھنے کے لئے کمپائیلر ہر مرتبہ ()main سے اپنا کنٹرول ٹرانسفر کرنے سے پہلے ایک snap کھینچ لیتا ہے اور اسے میموری میں محفوظ کر دیتا ہے۔اس تصور کو ہم PCB یا Process control block کہتے ہیں۔اس بلاک میں تمام انفارمیشن ہوتی ہے کہ فنکشن کس لائن سے call کیا گیا ہے اور اب کنٹرول نے کہاں ریٹرن ہونا ہے۔کمپائیلر ریٹرن ہونے سے پہلے PCB کو پڑھتا ہے، میموری سے Delete کرتا ہے اور بقیہ پروگرام مکمل کرتا ہے۔تصویر سے سمجھیں:



یہ تمام کمپائیلر کے کام کرنے کا طریقہ تھا۔ہم نے فنکشن کا کوڈ تو صرف ایک بار ہی لکھا مگر اس switching میں پروگرام کی speed بہت کم ہوگئی۔اب ہم چاہ رہے ہیں کہ نہ تو ہم فنکشن کی کوڈنگ بار بار لکھیں اور نہ ہی اسپیڈ میں کوئی فرق آئے۔تو اس صورت میں ہم ان لائن فنکشن استعمال کرتے ہیں۔

inline کا مطلب

inline فنکشن کو بھی ہم صرف ایک بار لکھتے ہیں مگر جہاں جہاں یہ پروگرام میں call ہوگا، کمپائیلر exe فائل میں اس پورے فنکشن کو وہاں کاپی کر دے گا تاکہ پروگرام sequentially چل سکے اور Switching میں وقت ضائع نہ ہو۔

inline فنکشن بناتے ہوئے اس بات کا خیال رکھیں کہ ان لائن فنکشن میں Prototype نہیں ہوتی بلکہ ہم مکمل فنکشن کو ()main سے پہلے لکھ دیتے ہیں کیونکہ یہاں کمپائیلر کو calling پر پورا فنکشن کاپی کرنا ہوتا ہے، نہ کہ صرف اس کا Address۔

Syntax

syntax کے اعتبار سے ان لائن فنکشن بہت آسان ہیں۔ عام فنکشن بنایئے اور فنکشن ٹائپ سے پہلے کی ورڈ inline لکھ دیجئے، باقی سب کچھ کمپائیلر خود سمجھ جائے گا۔calling کے وقت کچھ لکھنے کی ضرورت نہیں، وہ عام فنکشن کی طرح ہی ہوگی۔ایک بات پھر ذہن میں نوٹ کر لیجئے کہ inline فنکشن کا پہلا اور آخری مقصد پروگرام کی speed میں اضافہ کرنا ہے اور بس! آیئے، ان لائن فنکشن کا ایک پروگرام دیکھ لیں۔

جب آپ اس پروگرام کو کمپائل کر کے exe فائل کا size دیکھیں گے تو آپ کو بخوبی اندازہ ہو جائے گا۔

INLINE FUNCTION PROGRAM

```
#include <conio.h>
#include <stdio.h>
inline int add (int a ,int b)
{
return  (a+b);
}
inline int sub(int a, int b)
{
return  (a - b);
}
void main(void)
{
printf("Addition=%d",  sum(10,5));
printf("Difference=%d",sub(10,5));
getch();
}
```

اسکوپ ریزولوشن آپریٹر

اسکوپ ریزولوشن آپریٹر ہم کہتے ہیں ڈبل کولون سائن (::) کو۔اسکوپ scope کا مطلب ہوتا ہے پروگرام کا وہ حصہ جہاں ہم کسی ویری ایبل کو استعمال کر سکیں۔

مثال کے طور پر ایک یوزر ڈیفائن فنکشن ہے ABC، اس فنکشن کے اندر ایک ویری ایبل ہے ;int y۔اب آپ کیا اس y کو ()main میں استعمال کر سکتے ہیں؟ ہرگز نہیں کیونکہ ہر ویری ایبل لوکل ہے اور صرف اسی فنکشن میں قابلِ استعمال ہے جہاں ڈیفائن کیا گیا ہے بالکل اسی طرح ()main کے ویری ایبل کو ہم کہیں اور استعمال نہیں کر سکتے۔

ہم چاہ رہے ہیں کہ ایک ایسا ویری ایبل بنایا جائے جو global ہو اور اسے پروگرام کے کسی بھی حصے میں استعمال کیا جا سکے تو ہم صرف کیا کریں گے کہ ویری ایبل کو ()main سے پہلے Declare کر دیں گے۔اب آپ اس ویری ایبل کو پورے پروگرام میں استعمال کر سکتے ہیں۔دیکھئے، کس طرح:

GLOBAL VARIABLE PROGRAM

```
#include <conio.h>
#include <stdio.h>
int i;
void user(void)
{
printf("%d", i);
}
void main(void)
{
i=10;
user();
}
```

اس پروگرام میں ہم نے ایک ہی ویری ایبل کو main اور یوزر ڈیفائن دونوں فنکشن میں استعمال کیا ہے۔ مندرجہ بالا پروگرام میں ہم نے Pascal یا inline کے فنکشن کی طرح Function Definition پہلے دے دی ہے۔ یہ Syntax بھی درست ہے۔

اب سمجھئے کہ Scope Resolution آپریٹر کی ضرورت کہاں پیش آتی ہے۔

مثلاً آپ کے پاس دونوں Scopes میں ایک ہی نام کا ویری ایبل موجود ہے۔ اب آپ جب بھی ویری ایبل کو استعمال کرنا چاہیں گے تو لوکل ویری ایبل Priority یعنی سبقت لے جائے گا۔ تو کیا آپ گلوبل ویری ایبل کو استعمال نہیں کر سکتے۔ ان حالات میں اسکوپ ریزولوشن آپریٹر استعمال ہوتا ہے۔

اگر دونوں اسکوپ میں ایک ہی نام کا ویری ایبل موجود ہو اور آپ گلوبل ویری ایبل کی ویلیو کو استعمال کرنا چاہ رہے ہوں تو variable سے پہلے :: لکھ دیجئے، کمپائیلر سمجھ جائے گا کہ بات گلوبل ویری ایبل کی ہو رہی ہے۔ اب وہ لوکل کو Priority نہیں دے گا۔ آیئے پروگرام سے سمجھتے ہیں:

```
#include <conio.h>
#include <stdio.h>
int  i=10;
void main(void)
{
int i=5;
printf("local variable= %d", i);
printf("global variable= %d", ::i);
printf("sum  of variables= %d", ::i+i);
:: i=20; // change the value of global
i = 20; // change the value of local
}
```

بس، صرف اتنی سی بات یاد رکھیں کہ جب لوکل اور گلوبل اسکوپ دونوں میں ویری ایبل ایک ہی نام کا ہو تو ہم گلوبل ویری ایبل کی پہچان کے لئے اس کے ساتھ :: آپریٹر استعمال کرتے ہیں۔ Scope سے متعلق ہم تفصیلاً آبجیکٹ اورینٹڈ پروگرامنگ (OOP) کے حصے میں Storage classes کے تحت پڑھیں گے۔

ریفرنس آرگیومنٹ (Reference Argument)

سب سے پہلے تو آرگیومنٹ کو سمجھ لیں۔ آرگیومنٹ یا پیرامیٹر ایک ہی چیز کے دو نام ہیں۔ یہ آپ پڑھ چکے ہیں وہ units جن کی مدد سے کوئی فنکشن ویلیو بھیجتا یا وصول کرتا ہے، اسے ہم آرگیومنٹ کہتے ہیں۔ اب آیئے دیکھتے ہیں کہ ریفرنس کیا ہے۔

REFERENCE کا مطلب

ریفرنس کا مطلب ہوتا ہے Alias یا پہلے سے بنے ہوئے ویری ایبل کا دوسرا نام۔ پہلے آپ ریفرنس اور اس کا Syntax سمجھ لیں، بعد میں ہم بات کریں گے کہ ایسا ہم کیوں کرتے ہیں۔ سادہ اور آسان الفاظ میں اگر پہلے سے موجود ویری ایبل کو کوئی دوسرا نام دیا جائے تو وہ ریفرنس ویری ایبل کہلاتا ہے۔ ریفرنس ویری ایبل درحقیقت کچھ نہیں بجز اس کے کہ وہ جس ویری ایبل کا ریفرنس ہے، اس کی طرف نشاندہی کردے۔

Syntax

```
datatype &variable_name=variable;
```

ریفرنس ویری ایبل تخلیق کرتے ہوئے یہ بات ضروری ہے کہ آپ اسے کسی نہ کسی ویری ایبل کے ساتھ initialize کریں تاکہ پتا چل سکے کہ وہ کس کا ریفرنس ہے۔

چلیں پہلے آپ ایک پروگرام اور اس کا output دیکھیں پھر Reference پر کچھ اور بات کرتے ہیں۔

```
#include <conio.h>
#include <stdio.h>
void main(void)
{
int i=5;
int &ref=i;
printf("%d",   i);printf("%d",ref);
i++;  printf("%d",i);
ref++;  printf("%d",i);
ref=10;
printf  ("%d",i);
}
```

OUTPUT

```
5
5
6
7
10
```

سب سے پہلے ایک ویری ایبل تخلیق ہوا جس کی قیمت 5 ہے۔ اب ہم نے & یا اینڈ سائن کی مدد سے i کا دوسرا نام ref رکھ دیا۔ اب ہم خواہ i لکھیں یا ref بات ایک ہی ہے۔ ایک ہی میموری لوکیشن کے دو نام ہیں۔ ہم i میں اضافہ یا تبدیلی کریں یا ref میں، اثر دونوں پر ہوگا۔

```
        Memory
i/ref  --------> [5]
          IFF
```

ریفرنس کا استعمال

ریفرنس کو عموماً ہم فنکشن کے ساتھ استعمال کرتے ہیں۔ جب ہم فنکشن کی طرف کوئی Actual Argument پاس کرتے ہیں تو اتنی ہی جگہ یہ میموری میں پھر سے Formal Argument کی صورت میں لیتا ہے۔

مثال کے طور پر اگر ہم کسی فنکشن کی طرف کوئی ویلیو پاس کرتے ہیں تو فنکشن کے پاس اس ویلیو کو محفوظ کرنے کے لئے ایک الگ ویری ایبل ہوگا۔ چلئے ایک پروگرام سے سمجھتے ہیں:

```
# include <conio.h>
# include <stdio .h>
void  user(int);
void main(void)
{
int i=5;
user(i);
}
void user(int j)
{
j++;
printf  ("%d",j);
}
```

مندرجہ بالا پروگرام کو دیکھیں۔ i کی ویلیو main سے پاس ہوئی یوزر کی طرف؛ جہاں یوزر فنکشن کے ویری ایبل j میں اسے محفوظ کر دیا گیا۔ بعد ازاں j میں 1 ویلیو کا اضافہ کیا گیا اور j کی ویلیو پرنٹ ہوگئی۔ اب دو بائٹس main میں خرچ ہوئے اور 2 بائٹس مزید user میں۔ ہم چاہتے ہیں کہ صرف 2 بائٹس خرچ ہوں۔

ایک طریقہ تو یہ ہے کہ ہم ویری ایبل گلوبل بنا دیں۔ مگر اس صورت میں تو ویری ایبل کو کوئی بھی فنکشن استعمال کرے گا۔ ہم چاہتے ہیں کہ یہ اجازت صرف اور صرف main یا user کو حاصل ہو۔ دوسری بات یہ کہ کیا مندرجہ بالا پروگرام میں j++ سے i کی ویلیو میں کوئی فرق پڑا۔ جی نہیں! فنکشن کسی بھی طرح main کے لوکل ویری ایبل میں تبدیلی نہیں کر سکتا۔ اس صورتحال میں ہم پیرامیٹرز کو ریفرنس کی شکل میں پاس کریں گے تا کہ کوئی نیا ویری ایبل تخلیق نہ ہو۔ بلکہ صرف پہلے سے موجود ویری ایبل کا ایڈریس، فنکشن کی طرف پاس ہو جائے جسے ہم ریفرنس آرگیومنٹ کہتے ہیں۔ اس طرح ہم فنکشن میں رہتے ہوئے بھی main کے ویری ایبل کی ویلیو تبدیل کر سکتے ہیں کیونکہ پاس ہونے والا ویری ایبل تو صرف دوسرا نام ہے پہلے ویری ایبل کا۔ دونوں میں سے کسی پر بھی کام کرنے کی صورت میں ایک ہی میموری لوکیشن پر کام ہوگا۔ آیئے پروگرام سے سمجھتے ہیں۔

```
#include <conio.h>
#include  <stdio.h>
void user(int &);
void main(void)
{
int i=5;
user(i);
printf("%d",i);
}
void user(int &j)
{
j++;
}
```

پروگرام کا آؤٹ پُٹ آئے گا 6۔

جی ہاں! کیونکہ j صرف i کا دوسرا نام ہی تو ہے۔ جب ہم نے j میں اضافہ کیا تو مطلب ہے کہ i پر ہی کام ہو رہا ہے۔ جب ہم فنکشن کی طرف کوئی value پاس کرتے ہیں جو کہ میموری میں دوبارہ کاپی ہوتی ہے تو اس طریقے کو ہم call by value کہتے ہیں اور جب صرف ایڈریس پاس ہو یا اینڈ سائن & کی مدد سے ویری ایبل پاس ہو تو ہم اسے call by Reference کہتے ہیں۔ امید ہے کہ آپ کو ریفرنس سمجھ میں آ گیا ہوگا۔

ریفرنس کے فوائد

1- کم میموری کا خرچ ہونا۔

2- فنکشن میں رہتے ہوئے main کے اوریجنل ویری ایبل پر کام کرنا۔

3- فنکشن سے ایک سے زائد output ریٹرن ہونا۔

جی ہاں! ریفرنس کی مدد سے ہم فنکشن سے ایک سے زائد output ریٹرن کروا سکتے ہیں کیونکہ جتنی بھی ویلیوز & سائن کے ساتھ پاس ہونگی وہ براہ راست متاثر ہوں گی۔ اچھا، اب آپ جلدی سے مندرجہ ذیل پروگرام کا output بتا دیجئے:

```
#include <conio.h>
#include <stdio.h>
void swap(int,int);
void main()
{
int i=10,j=10;
swap(i,j);
printf("After  swapping");
printf("\n j is = %d",j);
printf("\n i is = %d",i);
}
void swap(int a, int b)
{
int temp;
temp=a;
a=b;
b=temp;
}
```

اس پروگرام کا output یہ آئے گا:

```
After swapping
j  is=20
i  is=10
```

جی ہاں swapping نہیں ہوگی کیونکہ ویلیوز ہم main() سے پرنٹ کروا رہے ہیں اور جو کچھ بھی عمل ہوا ہے وہ a اور b فنکشن کے formal آرگیومنٹس پر ہوا ہے۔ i اور j کا ان سے کوئی تعلق نہیں۔ اگر آپ چاہتے ہیں کہ پروگرام درست کام کرے تو صرف اتنا کریں کہ پروٹو ٹائپ اور definition پارٹ میں ڈیٹا ٹائپ کے بعد & لگا دیجئے۔

```
void swap(int&, int&);
void swap(int& a, int& b)
{
:
}
```

آپ کا پروگرام درست ہو گیا۔

آیئے، باب کے آخر میں فنکشن سے متعلق چند اہم باتیں نوٹ کرلیں۔

1- سی (C) کا ہر پروگرام ایک یا ایک سے زائد فنکشن کا مجموعہ ہے۔

2- فنکشن تب کال ہوتا ہے جب آپ فنکشن کا نام اور اس کے بعد سیمی کولن لکھتے ہیں۔ مثلاً ;()user

3- فنکشن میں ہم فنکشن کے نام کے بعد کرلی بریکٹس {} میں ایک یا ایک سے زائد statements لکھتے ہیں۔

4- کسی بھی فنکشن کو ہم کسی اور فنکشن سے کال کر سکتے ہیں حتیٰ کہ ()main کو بھی کال کر سکتے ہیں۔

5- وہ طریقہ کار جس میں ایک فنکشن اپنے آپ کو ہی کال کرتا ہے، Recursion کہلاتا ہے جو ہم آئندہ ابواب میں پڑھیں گے۔

6- ایک فنکشن کو ہم جتنی مرتبہ چاہیں کال کر سکتے ہیں۔ تعداد پر کوئی پابندی نہیں ہے۔

7- جس ترتیب سے ہم نے فنکشن کو ڈیفائن کیا ہے، ضروری نہیں کہ اسی ترتیب سے کال بھی کریں۔ call کرتے ہوئے ترتیب کچھ بھی ہو سکتی ہے۔

8- ایک فنکشن کو ہم کسی بھی دوسرے فنکشن سے کال تو کر سکتے ہیں مگر ڈیفائن نہیں کر سکتے Definition part ہر صورت میں فنکشن کے باہر ہوگا۔

9- فنکشن بنیادی طور پر صرف دو طرح کے ہوتے ہیں۔

یوزر ڈیفائن جو ہم خود لکھتے ہیں اور لائبریری فنکشن جو کمپائیلر کے ساتھ آتے ہیں۔ یہ فنکشن کس نے لکھے ہوتے ہیں؟ جس نے کمپائیلر لکھا ہوتا ہے۔ لائبریری فنکشن بہت زیادہ استعمال ہونے والے فنکشن کا مجموعہ ہوتے ہیں جو کہ یوزر کی آسانی کے لئے پہلے سے لکھ دیئے گئے ہوتے ہیں۔

بہر طور، دونوں طرح کے فنکشن کی calling کا طریقہ کار بالکل یکساں ہے۔ تو جناب! فنکشن ختم ہوئے۔ ان کے علاوہ بھی فنکشن کی مختلف صورتیں ہو سکتی ہیں جیسے:

Member Functions

Inline Members

Virtual Functions

Pure Virtual Function

Friend Function