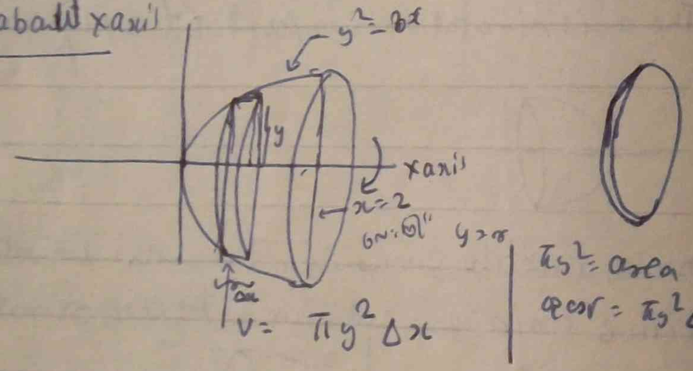
③ Assume the rectangle assumed the indefinitely increased

④ Apply the fundamental theorm

(24) Example (1)

Find the volume generated by revolving the first quadrant area bounded by the parabola $y^2 = 8x$ and it lathe lectam ($x = 2$) about the x axis.

(a) rotate about x axis



$V = \pi y^2 \Delta x$

πy^2 = area

area $= \pi y^2 \Delta x$

For all volume

$$V = \sum_{n=1}^{\infty} \pi y^2 \Delta x = \sum_{n=1}^{\infty} \pi y^2 dx$$

$$= \int_a^b \pi y^2 dx$$

$a = 0$, $x = 2$

$\therefore a = 0$ ලෙස, $b = 2$ ලෙස නියම

$$V = \int_0^2 \pi y^2 dx$$

$$V = \pi \int_0^2 8x \, dx$$

$\Delta x : dx \to 0$

$x = y^2 = 8x$

$x = 0 : y = 0$

$x = 2$ නම් $y = \sqrt{8 \times 2} = 4$

$\therefore$ $0 \le y \le 4$ ලෙස : නියම $a = 0$ $b = 4$

$$V = \int_0^4 \pi (2 - x)^2 dy$$ { නියම

$$V = \int_0^4 2\pi (2 - x)^2 dy$$ නියම

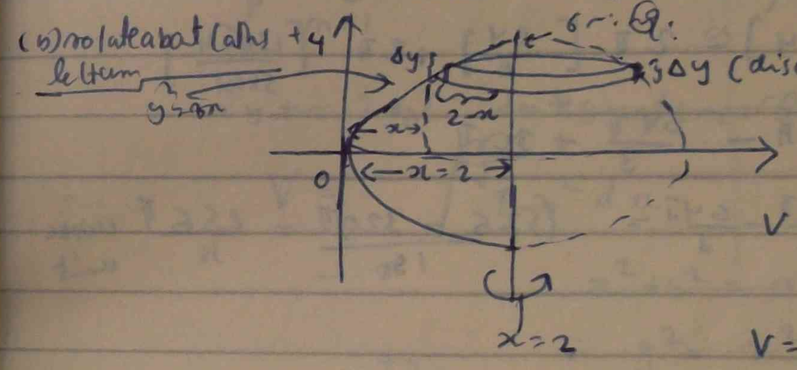
$$= 2\pi \int_0^4 (2 - \frac{y^2}{8})^2 dy$$

$$V = 4\pi x \Big|_0^2$$

$$= 16\pi \text{ square unit}$$

when rotate about the lathe lectam to find volume

(b) rotate about lathe lectam



$$V = \sum_{k=1}^{\infty} \pi (2 - x)^2 \Delta y$$

$$V = \sum_{k=1}^{\infty} \pi (2 - x)^2 dy$$

k = indefinitely increases නම් integral එක

$$V = \int_a^b \pi (2 - x)^2 dy$$

$$V = 2\pi \int_0^4 \left[4 - \frac{y^2}{4} + \frac{y^4}{64}\right] dy$$

$$= 2\pi \cdot 4[y]_0^4 - 2\pi\left[\frac{y^3}{6}\right]_0^4 + 2\pi\left[\frac{y^5}{5\times 64}\right]_0^4$$

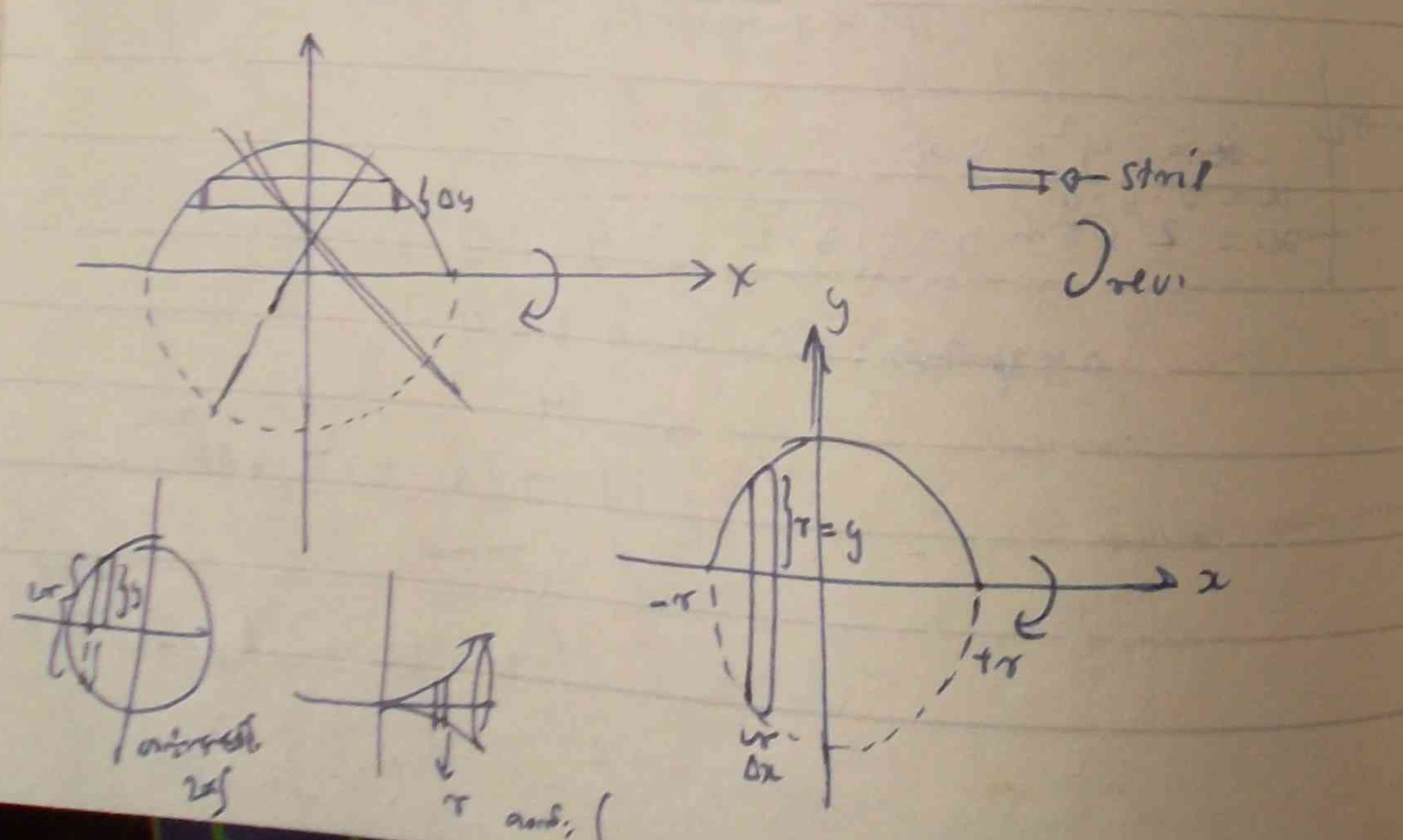
$$= 8\pi[4] - 2\pi\left[\frac{64}{6}\right] + 2\pi\left[\frac{256\times 4}{5\times 64}\right]$$

$$= 32\pi - \frac{64\pi}{3} + \frac{32\pi}{5}$$

$$= \frac{192\pi}{5} - \frac{64\pi}{3} = \frac{(576-320)\pi}{15} = \frac{256\pi}{15}$$ cubic unit

Example ②

Find the volume of a sphere by revolving an area of semi circle



$$V = \pi y^2 \Delta x$$

$$V = \sum_{n=1}^{\alpha} \pi y^2 \Delta_n x$$

$$= \sum_{n=1}^{\alpha} \pi y^2 dx = \int_a^b \pi y^2 dx$$

$-r$ မှ $+r$ ထိ Integrate လုပ်မယ်

$$V = \int_{-r}^{r} \pi y^2 dx$$

$$x^2 + y^2 = r^2$$

$$\therefore y^2 = r^2 - x^2$$

$$V = \int_{-r}^{+r} \pi \{r^2 - x^2\} dx$$

$$= \int_{-r}^{+r} \pi r^2 dx - \int_{-r}^{+r} \pi x^2 dx$$

$$= \left[\pi r^2 x\right]_{-r}^{r} - \left[\pi \frac{x^3}{3}\right]_{-r}^{r}$$

$$= \pi r^2 [r - (-r)] - \frac{\pi}{3}[r^3 - (-r)^3]$$

$$= \pi r^2 [2r] - \frac{\pi}{3}[2r^3]$$

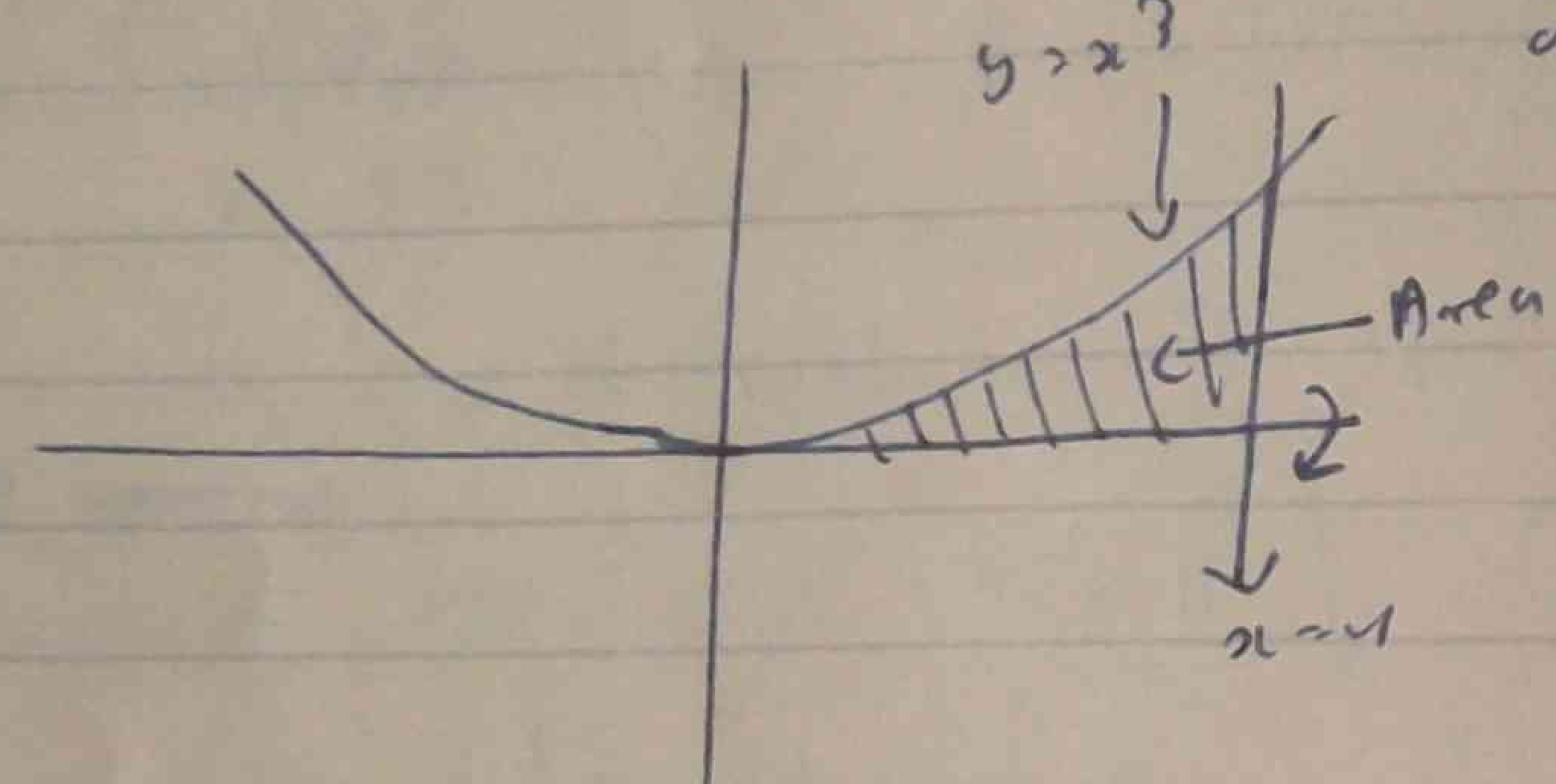
$$= 2\pi r^3 - \frac{2\pi}{3} r^3$$

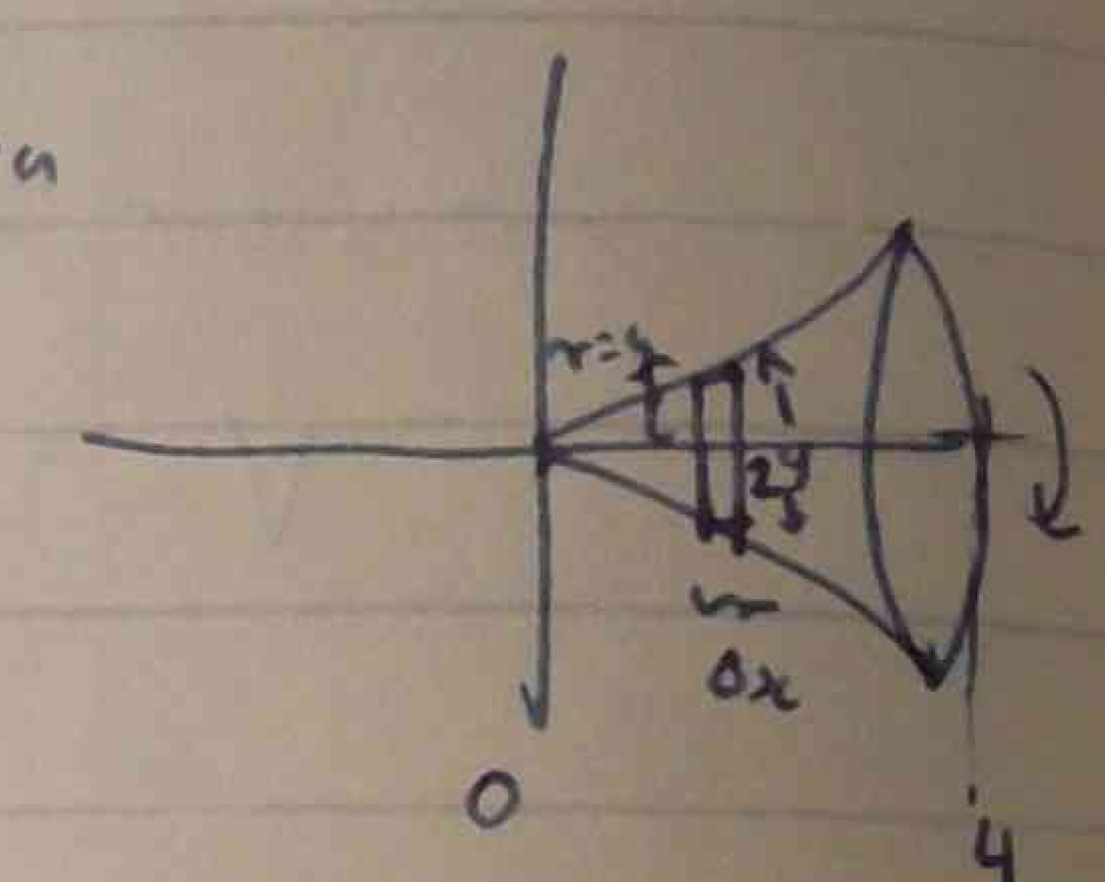
$$= \frac{4}{3}\pi r^3$$

(26)

~~Example~~ Exercise ①

Find the volume of generated by revolving the first quadrant area bounded by the curve $y = x^3$ and the line $x = 4$ about x axis





$$V = \frac{2}{3}\pi r^2 \Delta x = \pi y^2 \Delta x$$

$$\sum_{n=1}^{\infty} \pi y^2 \Delta_n x$$

$$\text{for full} = ~~2~~ \left[\sum_{n=1}^{\infty} \pi y^2 \Delta_n x\right]$$

$$= ~~2~~ \int_0^4 \pi y^2 \, dx$$

$y = x^3 \quad \therefore y^2 = x^6$

$$= ~~2~~ \int_0^4 \pi x^6 \, dx = ~~2~~\pi \left[\frac{x^7}{7}\right]_0^4$$

$$= ~~2~~\pi \left[\frac{4^7}{7}\right]$$

$$= ~~2~~\pi \times \frac{4096}{7} = \frac{8192\pi}{7} = 1170.2\ldots \text{ cubic units}$$

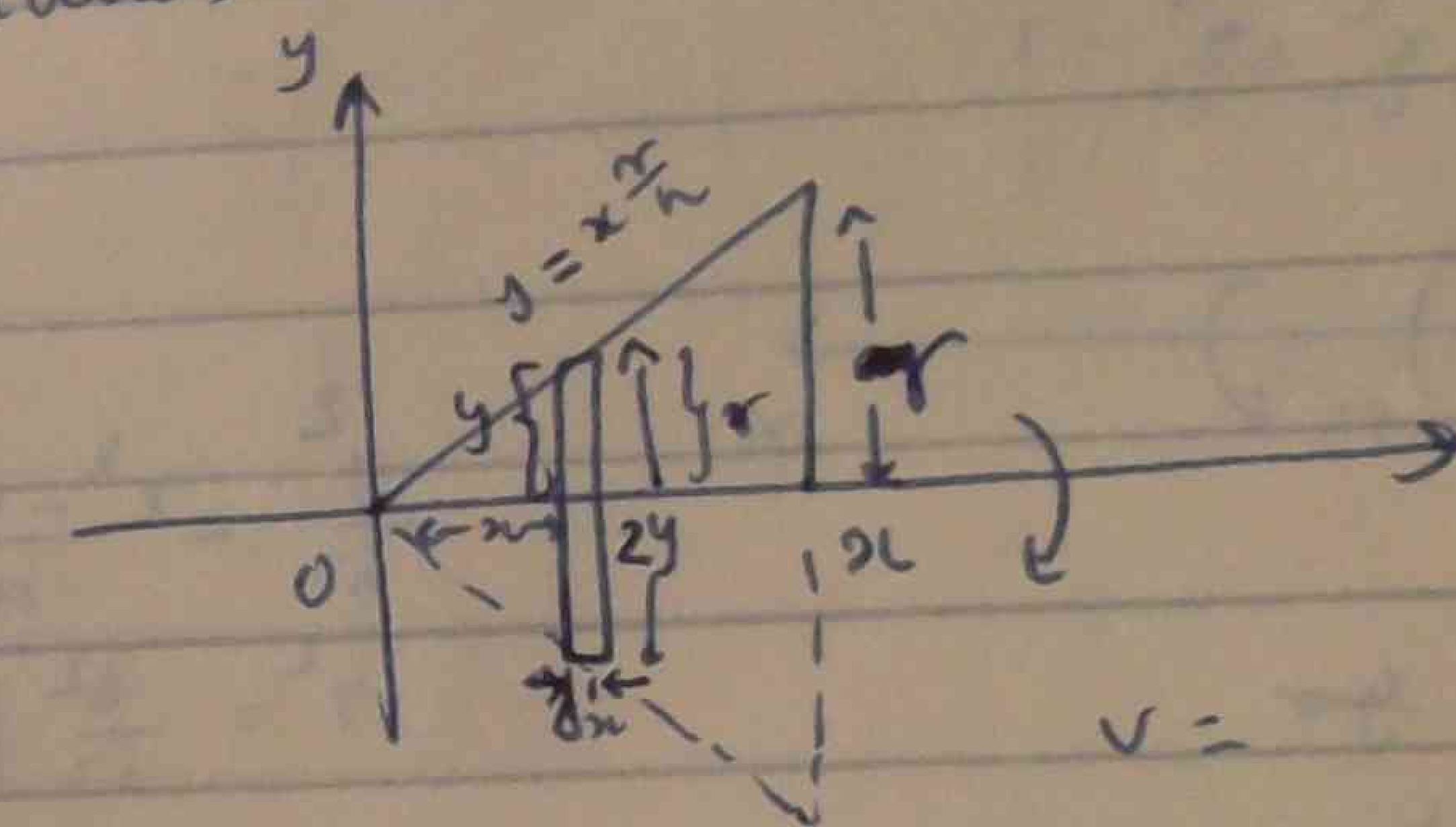
~~axis ... $\int \pi y^2 dx$~~

~~axis ... $\int \pi y^2 dx$~~ [illegible]

(27)

~~Example~~ Exercise ③

Find the volume of cone by applying Integral calculus



$0 \to \frac{x}{h} \quad \therefore 0 \to h$

$$V = \sum_{n=1}^{\infty} \pi y^2 \Delta_n x$$

$y = r$

$$= \int_0^h \pi y^2 \, dx \quad \text{—— ①}$$

$$\frac{y}{r} = \frac{x}{h} \quad \therefore y = \frac{xr}{h}$$

$$= ~~\int_0^h \pi x^2 dx~~$$

$$y^2 = \frac{x^2 \times r^2}{h^2}$$

$$V = \int_0^h \pi \frac{(x^2 \times r^2)}{h^2} \, dx$$

$$= \pi \frac{r^2}{h^2} \int_0^h x^2 \, dx = \frac{\pi r^2}{h^2} \times \frac{x^3}{3}\Big|_0^h = \frac{\pi r^2}{h^2} \times \frac{h^3}{3}$$

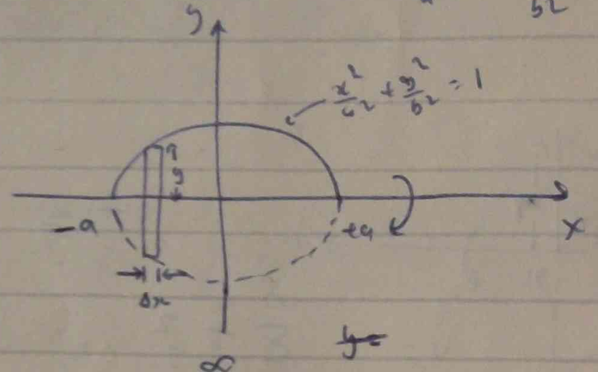
$$= \frac{1}{3} \pi r^2 h$$

axis of ... [illegible]

axis ... [illegible]

Exercise

(29) (2) Find the volume of the ellipse formed by revolving the positive half of the ellipse $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$ about the x axis.



$$\frac{y^2}{b^2} = x^2 - \frac{a^3}{a^2}$$

$$y^2 = \frac{b^2}{a^2}[x^2 - a^2]$$

$$V = \lim_{n\to\infty} \sum_{n=1}^{a} \pi y^2 \Delta x$$

$$= \int_{-a}^{a} \pi y^2 \, dx$$

$$= \int_{-a}^{a} \pi \frac{b^2}{a^2}[x^2 - a^2] \, dx$$

$$= \left[\int_{-a}^{a} \pi \frac{b^2 x^2}{a^2} dx - \int_{-a}^{a} \pi \frac{b^2 a^2}{a^2} dx \right]$$

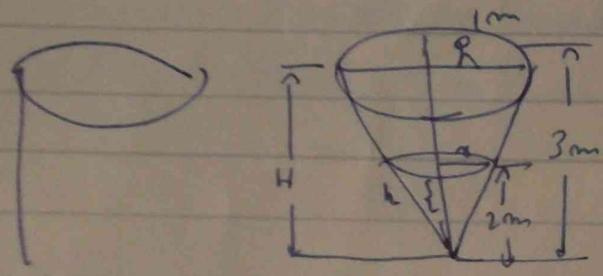
$$= \left[\pi \frac{b^2}{a^2} \left[\frac{x^3}{3}\right]_{-a}^{a} - \pi b^2 [x]_{-a}^{a} \right]$$

$$= \left[\frac{\pi b^2}{a^2} \left[\frac{a^3}{3} - \frac{a^3}{3}\right] - \pi b^2 [a^2 + (-a)^2] \right]$$

$$= 2[a^2 b^2 \pi] = 2a^2 b^2 \pi$$

Revision For Final Exam

(12)



$\frac{dv}{dt} = 1200 cm^3/sec$

$\frac{dr}{dt} = ?$

$V = \frac{1}{3}\pi r^2 h$

$\frac{R}{r} = \frac{H}{h}$ $r =$

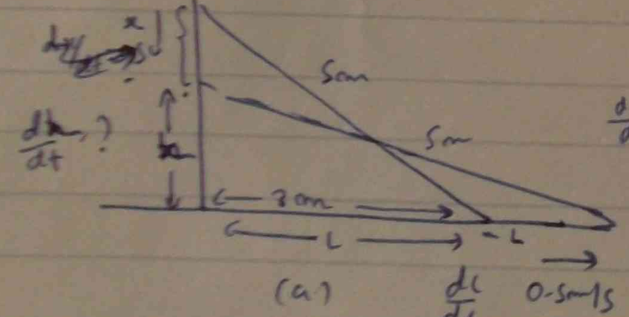
$\frac{dV}{dr} = \frac{dv/dt}{dr/dt}$

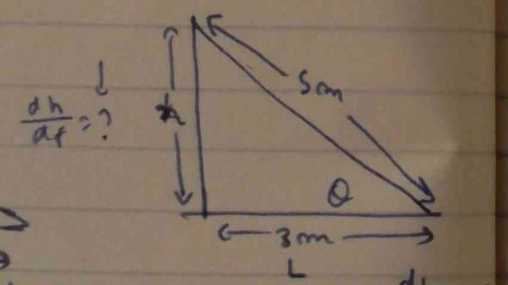
$\frac{1}{r} = \frac{3}{h}$

$\frac{d\,\frac{1}{3}\pi r^2(\frac{1}{3}r)}{dr} = \frac{1200}{dr/dt}$

$\frac{dr}{dt} = -$ $h = \frac{1}{3}r$

(13)





$s^2 = h^2 + L^2$ — (1)

$\frac{ds^2}{dt} = \frac{dh^2}{dt} + \frac{dL^2}{dt}$

$0 = 2h\frac{dh}{dt} + 2L\frac{dL}{dt}$

$0 = 2 \times h \times \frac{dh}{dt} + 2 \times 3 \times 0.5$

$0 = 2 \times 4 \frac{dh}{dt} + 3$

$\therefore dh/dt = -\frac{3}{8}$ m/s

at: of $h = \sqrt{5^2 - 3^2} = 4$

(b)

$h = L\tan\alpha$

$\frac{dh}{dt} = \frac{dL\tan\alpha}{dt}$ | $\frac{dh}{dt} = L$

$\tan\alpha = \frac{h}{L}$

$\frac{d\tan\alpha}{dt} = \frac{d\frac{h}{L}}{dt} = \frac{h^2\frac{dL}{dt} - L^2\frac{dh}{dt}}{L^2}$

$\frac{dslope}{dt} = \frac{4^2 \times \{-\frac{3}{8}\} - 3^2 \times \{0.5\}}{3^2} = \epsilon$

(14)

Mathematics

Revision for Final Exam

~~stat~~

Statistics

① Range = 176 − 119 = 57

No of class interval × class interval size = 57

6 × 9 ≃ 57

class interval size − 1 = 9 − 1 = 8 //

$\frac{\text{upper limit} + \text{lower limit}}{2} = 119$

upper + lower = 119 × 2 = 238

upper + lower − 8 = 238 − 8 = 230

∴ class mark = $\frac{230}{2} = 115$ ← Boundary Boundary

115 → (115+8) = 115 → 123

124 → (124+8) = 124 → 132

133 → (133+8) = 133 → 141

142 → (142+8) = 142 → 150

151 → (151+8) = 151 → 159

160 → (160+8) = 160 → 168

169 → (169+8) = 169 → 177

class	class mark x_j	frequency f_j	$f_j x_j$	d_j $x_j - A$	a_j	$f_j a_j$	$(x_j - \bar{x})^2$
115 → 123	119	1	119	−17	−3.87		
124 → 132	128	4	512	−18	−2.57		
133 → 141	137	8	1096	−9	−1.25		
142 → 150	146	14	2044	0	0		
151 → 159	155	8	930	9	+1.25		
160 → 168	164	5	820	18	+2.57		
169 → 177	173	2	346	27	3.87		

69 //

$d_j = a_j - A$

$a_j = d_j / c$

c = 115.5 ~~f~~. 122.5 ~~8~~. ~~f~~. ~~st~~

c = 122.5 − 115.5 = 7

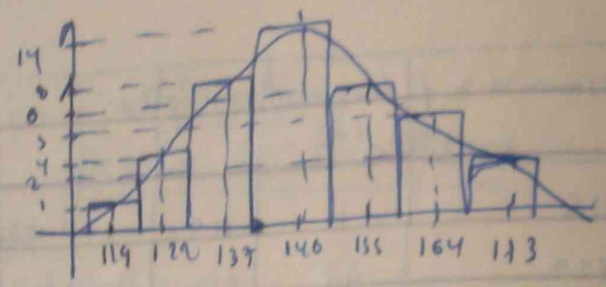
long method: $\bar{x} = \frac{\sum_{j=1}^{n} f_j x_j}{\sum_{j=1}^{n} f_j}$

short method $\bar{x} = A + \frac{\sum_{j=1}^{N} f_j d_j}{N}$

coding method $\bar{x} = A + \left(\frac{\sum_{j=1}^{n} f_j u_j}{N}\right) c$

$s = \sqrt{\frac{\sum_{j=1}^{n} (x_j - \bar{x})^2}{N}}$

(a)



$\bar{x}+s$, $\bar{x}-s$ = 68.27%

$\bar{x}+2s$ $\bar{x}-2s$ 95.45%

$\bar{x}+3s$ $\bar{x}-3s$ 99.73%

} $x_i : f_i$ (11)

common Logarithm

(2)

$(0.06626)^{1/1.4}$

$N = (0.06626)^{\frac{1}{1.4}}$

$\log N = \frac{1}{1.4} \log(0.06626)$

$= \frac{1}{1.4}[\bar{2}.8213]$

$= \frac{1}{1.4}[-2 + 0.8213]$

$= \frac{1}{1.4}[-1.1787]$

$= -0.8416$

$\log N = \bar{1} + 1 - 0.8416$

$= \bar{1}.1584 =$

$N =$ Antilog $\bar{1}.1584 = 0.144$ //

1.1787	0.0812
1.4	0.1461
0.8416	$\bar{1}.9251$

Neperian logarithm

(3)

$N = \frac{233.6^{-1/2}}{0.07541^{-0.4}} = ?$

$N = \frac{0.07541^{0.4}}{233.6^{1/2}}$

$\ln N = 0.4 \ln 0.07541 - \frac{1}{2} \ln 233.6$

$= 0.4 \times \ln[7.541 \times 10^{-2}] - \frac{1}{2} \ln[2.336 \times 10^{2}]$

$= 0.4[2.0203 + \bar{5}.3743] - \frac{1}{2}[0.8033 + 4.4683]$

$= 0.4[2.0203 + 5 + 0.3743] - \frac{1}{2} \times 5.4683$

$= 0.4 \times (-2.5854) - 2.7342$

$= -1.03416 - 2.7342$

$= -3.77821$

$[-3.77821 - [\bar{5}.3743]]$

$[-3.77821 + 5.3743] = 1.6566$

$\therefore -3.7821 = 1.6566 + \bar{5}.3743$

$\therefore \ln N = [1.6566 + \bar{5}.3743]$

$= 5.241 \times 10^{-2} = 0.05241$ //

Solving the equations

④

$x \times 8 = 7 \times y$

$8x - 7y = 0$ —— ①

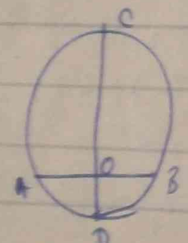
$(x-4) \times 6 = (y-4) \times 5$

$6x - 24 = 5y - 20$

$6x - 5y = 4$ —— ②

⑤ ~~Trigo~~ | Mensuration

(i)

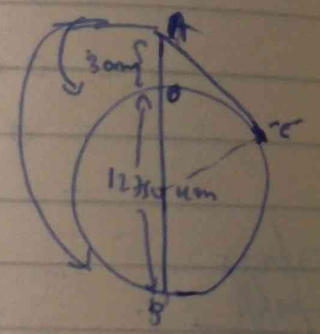


$AO \times BO = CO \times DO$

$AC^2 = AO \times AB$

$A = \frac{\theta}{360} \pi r^2$

(ii)



⑥ Trigo

$\sin(A \pm B) = \sin A \cos B \pm \cos A \sin B$

$\cos(A \pm B) = \cos A \cos B \mp \sin A \sin B$

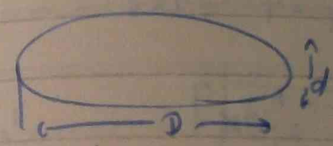
$\tan(A \pm B) = \frac{\tan A \pm \tan B}{1 \mp \tan A \tan B}$

$\frac{\theta}{360} \pi r^2$

$180° = \pi$ rad

360

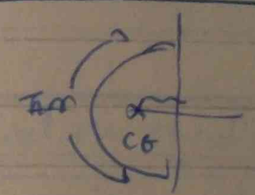
sector area $= \frac{\theta r^2}{2}$ θ in radian:



$A = \frac{\pi}{4} D \times d$

⑦ Theorem of Pappus

(i)



$l = \pi \times d/2$ $l = (\text{on}) \pi r$

dist: CG moved $= (\pi \times d) \times 2\pi x$

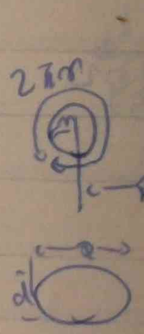
$= (2\pi x) \times (2\pi r)$

when $x = \frac{d}{\pi} = \frac{2r}{\pi}$

$\therefore$ $SA = 2\pi \times \frac{2r}{\pi} \times 2\pi r$

$= 4\pi r^2$ ※

(ii)



volume = length × dist: CG moved

$= 2\pi r \times [R \times 2\pi]$

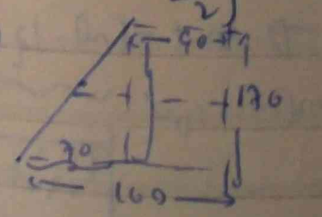
$= 4\pi^2 R r_2$

$= \pi(\frac{D+d}{2}) \times 2\pi R = \pi R (D+d)$

$= \pi^2 R (D+d)$

[length $= \pi \frac{(D+d)}{2}$

(iii)



length $= \sqrt{70^2 + 170^2}$

dist: CG moved $= (35 + 90) \times 2\pi$

$\sqrt{70^2 + 170^2} \times 2\pi (35 + 90)$

(8) **Simpson's Rule**

$A = \frac{d}{3}[1^{st} + 4\times 1^{nd} + 2\times 3^{rd} + 4\times 4^{th} + 2\times 5^{th} - - - last]$

$A = \frac{h}{3}[A_1 + 4A_2 + 2A_3 + 4A_4 - - - A_{last}]$

1, 3, 5, 7 ... odd

(9) **Differential calculus**

(i) limits

(a) $s_1 = 4.9t^2$

$s = 4.9(t+\delta t)^2$

$= 4.9(t^2 + 2t\delta t + \delta t^2)$

$\delta s = s_2 - s_1 = 4.9[t^2 + 2t\delta t + \delta t^2] - 4.9t^2$

$= 9.8t\delta t + 4.9(\delta t)^2$

$\delta s = 9.8\times 3\times(0.5) + 4.9(0.5)^2$

$=$

(b) $s = t^3 - 6t^2 + 9t + 4$

$\frac{ds}{dt} = 3t^2 - 12t + 9$

$v = 3t^2 - 12t + 9 \rightarrow a$

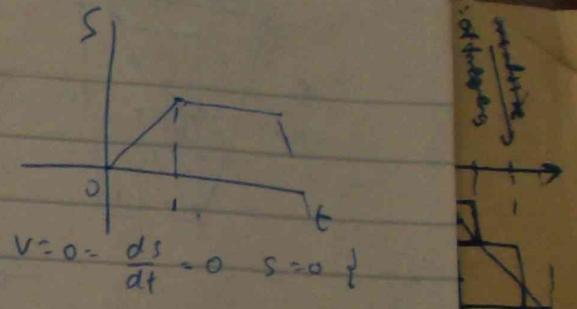
$t = (-)^{ve}(-)$ s → ... a ...

a ...

t_1 & t_2 ...: const

$t = 3 \quad a = 6, \quad s = 8$

$t = 1 \quad a = -6 \quad s = 8$



t_1 & t_2 ...

(t_1 and t_2) → ...

$t_1 \rightarrow t_2$ → ...

(10) **Application of calculus**

(i)

$\frac{dv}{dt} = 900 cm^3/s$

$\frac{dA}{dt} = ?$

$V = \frac{4}{3}\pi r^3$

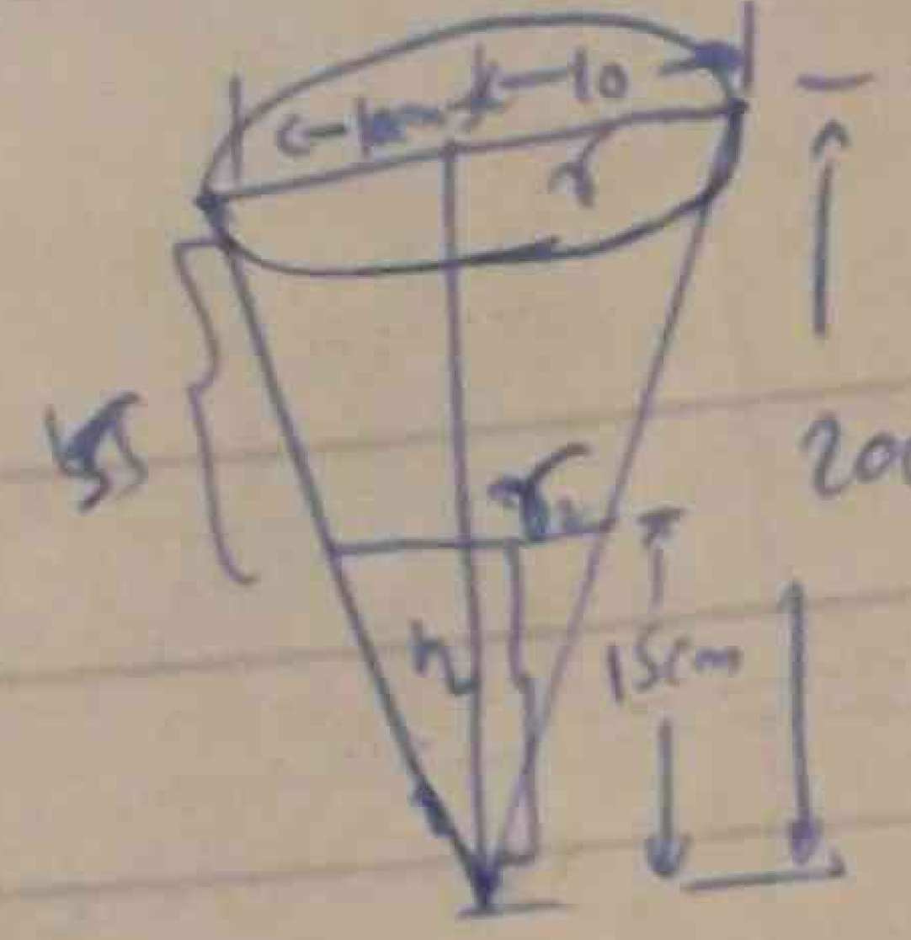
$\frac{dv}{dt} = \frac{dv}{dr}\cdot\frac{dr}{d}$ $\quad \frac{dv}{dA} = \frac{\frac{dv}{dt}}{\frac{dA}{dt}}$

$\therefore \frac{dA}{dt} = \frac{dv/dt}{dv/dA} = \frac{900}{d\,4/3\pi r^3 / d\,4\pi r^2} = \frac{900}{4\pi r^2 / 8\pi r}$

$= \frac{900}{1/2\,r} = \frac{1800}{r}$

when 360 $= \frac{1800}{360} = 5$

conical funnel



$$\frac{dV}{dt} = 5\,cm^3/s$$

$$dh/dt = ?$$

$$\frac{dV}{dh} = \frac{dV/dt}{dh/dt}$$

$$\frac{d[\tfrac{1}{3}\pi r^2 h]}{dh} = \frac{dV/dt}{dh/dt}$$

$$\frac{h}{H} = \frac{R}{r}$$ ~~r = ... h 5cm~~ R and H

~~...~~ r and h

$$\therefore \frac{r}{h} = \frac{R}{H}$$

$$h = \frac{H}{R} r = \frac{20}{10} r$$

$$h = 2r$$

$$\therefore \frac{d[\tfrac{1}{3}\pi r^2 \times \{2r\}]}{d[2r]} = \frac{5}{\frac{dh}{dt}}$$

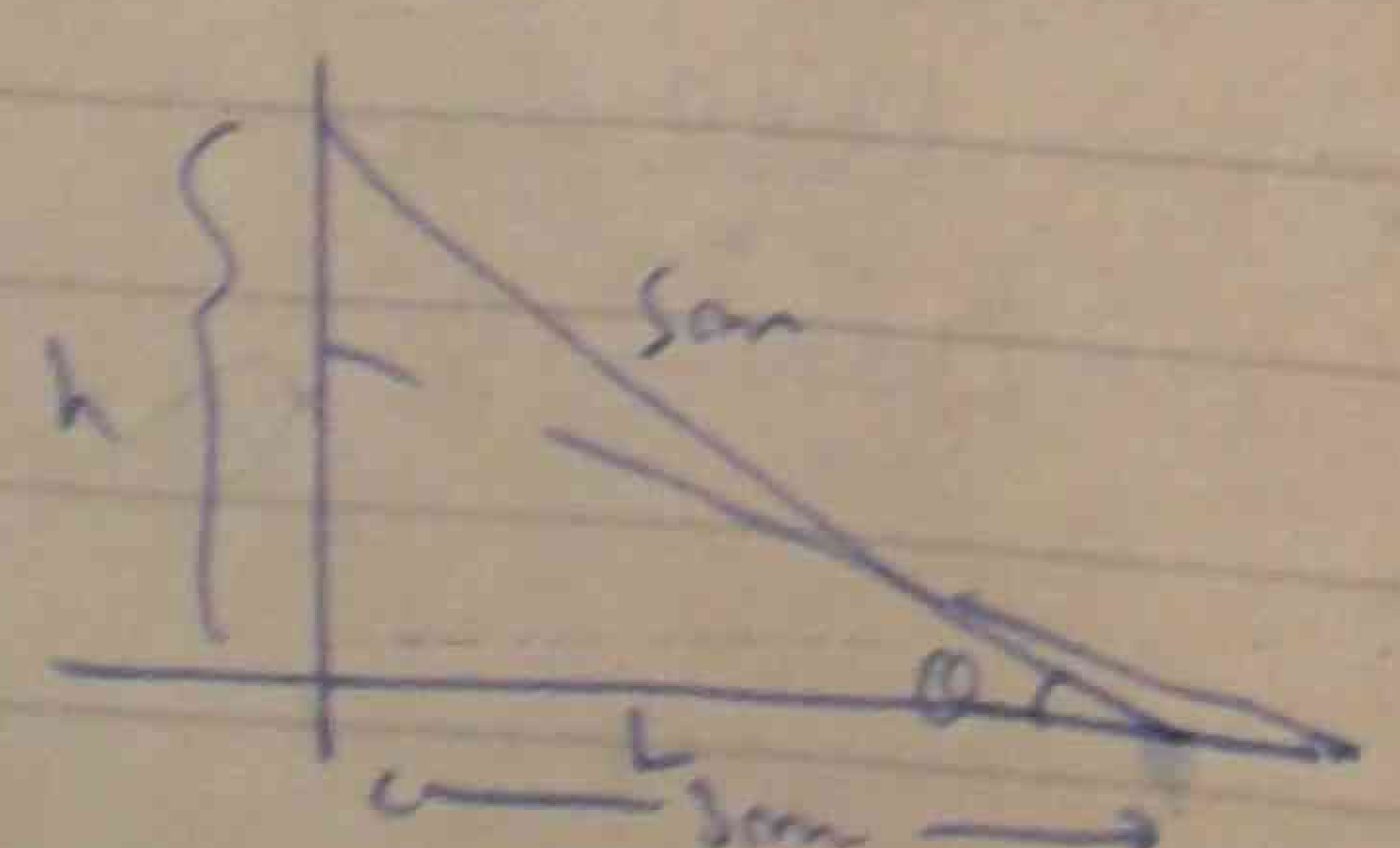
$$\frac{r}{15} = \frac{10}{20}$$

$$dh/dt = \frac{5 \times 2}{2\pi r^2 x}$$

$$r = 15/2$$

when r = ? h = 15cm r = $\sqrt{\ }$

(r = 15/2 ...)



$$h^2 + L^2 = 5^2 \quad \text{--- (1)}$$

$$h = L\tan\theta$$

slope

$$dh = dL\tan\theta$$

$$h^2 + L^2 = s^2$$

$$dh^2 + dL^2 = ds^2$$

~~d~~ $2h\,dh = -2L\,dL$

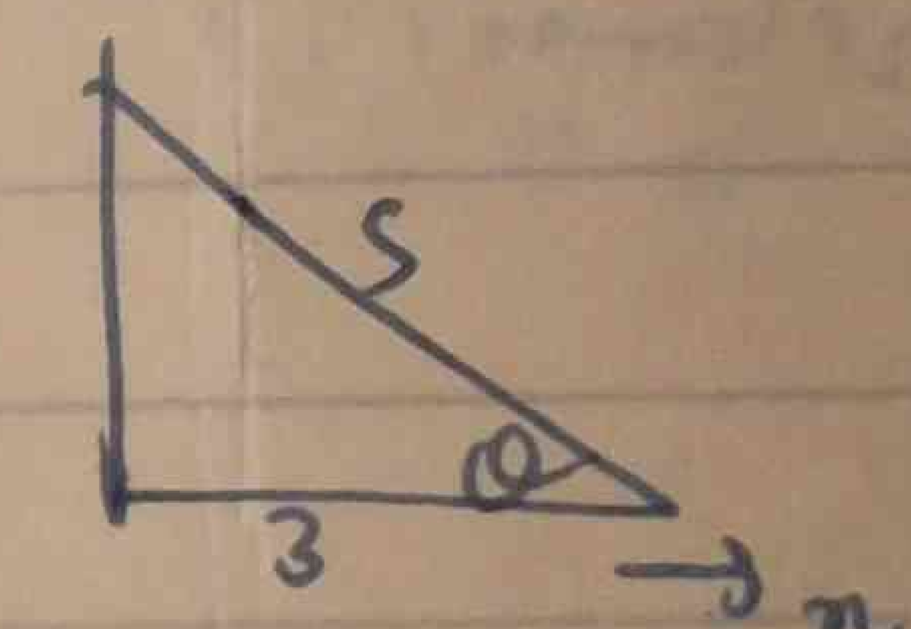
$$\therefore \frac{dh}{dL} = -\frac{L}{h}$$

when $L = 3$ $h = \sqrt{5^2 - 3^2}$

$$\therefore \frac{dh}{dL} = -\frac{3}{4} \quad \text{--- (1)}$$

$$\frac{dh}{dL} = \text{d slope}$$

$$\therefore \frac{d\,\text{slope}}{dt} = \frac{dh}{dL}$$ slope



$$\frac{dx}{dt} = 0.5\,m/s$$

~~d h/d~~

$$\frac{d\tan\theta}{dt} = ?$$

$$x\tan\theta = h$$

$$\frac{dx}{dt}\ d\,x\tan\theta = dh$$

(11) **max: and min:**

1st der. method — find max. value

$y = $ ~~$x^4 + x^3 - 3x^2 + x + 4$~~ $x^3 - 6x^2 + 9x - 8$

$$\frac{d\,x^3 + 6x^2 + 9x - 9}{dx} = 3x^2 - 12x + 9$$

$3x^2 - 12x + 9 = 0$ ... $x = 3$ and 1

~~x = 3 ... 1 ...~~ $x = 1 \longrightarrow 0,\ 2$ $\quad x = 3 \rightarrow 2,\ \rightarrow 4$

... dy/dx ... d

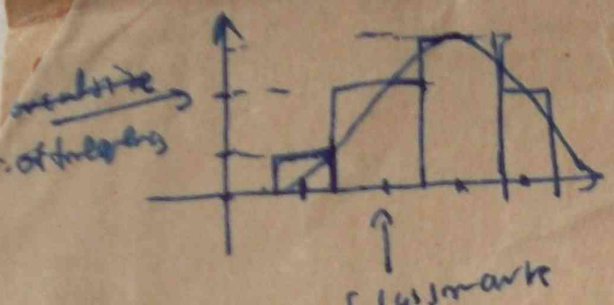
$$3(0)^2 - 12(0) + 9 = 9$$
$$3(1)^2 - 12(1) + 9 = -3$$

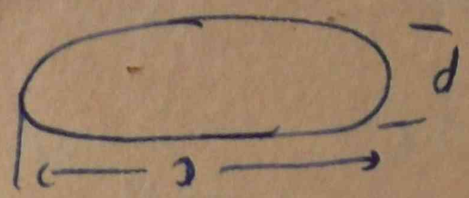
+ → − min at 0

$$3(2)^2 - 12(2) + 9 = (-)$$
$$3(4)^2 - 12(4) + 9 = +$$

− → + {max = } at 3 //

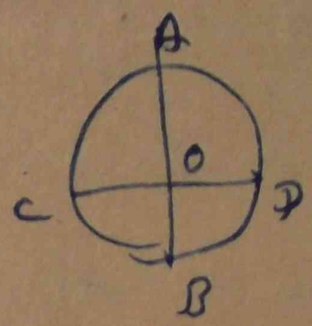
2
9
4
16





Area = $\frac{\pi}{4} D d$

circumference = $\frac{\pi}{2}(D+d)$



$AO \times BO = CO \times DO$

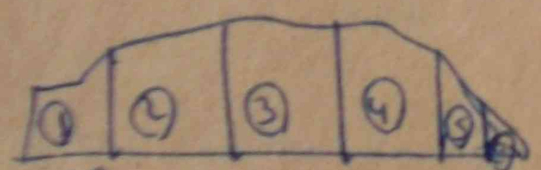
$AO \times BO = CO^2$

Theorem of Pappus

Surface Area = length × distance moved by CG

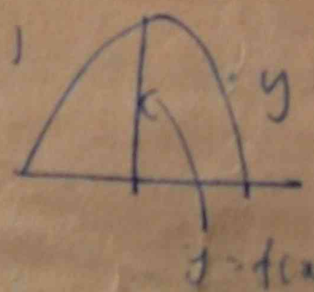
$x = \frac{d}{\pi}$

Area = $\frac{d}{3}[1^{st} + 4 \times 2^{nd} + 2 \times 3^{rd} + 4 \times 4^{th} \dots \text{last}]$



volume = $\frac{h}{3} \times \frac{\pi}{4}\{d_1^2 + 4 \times d_2^2 + 2 \times d_3^2 \dots \text{last}\}$

(Space part of suffix: (even)
ordinate suffix: (odd))



$y = f(x)$ area of suffix

Statistics

Formulases

(1) Data [illegible] = Range

(2) Number of class interval → [illegible]

$\{5 < x < 20\}$ × class interval size = range

(3) $$\frac{\text{lower} \times 2 - \{\text{class interval size} - 10^{-n} \times 1\}}{2} = \text{class } \text{one}$$

(4) class one (A) ⟶ [class one + (class interval size − $10^{-n} \times 1$)] (B)

[class one + (class interval size − $10^{-n} \times 1$)] + $10^{-n} \times 1$ (B+1) ⟶ [class one + (class interval size − $10^{-n} \times 1$)] + $10^{-n} \times 1$ + (class interval size − $10^{-n} \times 1$) (C)

(5) A ⟶ B, (B+1) ⟶ C } [illegible]

class frequency

(6)

Class	x_j class mark	frequency						
A ⟶ B	$\frac{A+B}{2}$							
(B+1) ⟶ C	$\frac{(B+1)+C}{2}$							

{ N = Total class }

(7) Long method $\bar{x}$ (mean) = $$\frac{\sum_{j=1}^{u} f_j x_j}{\sum_{j=1}^{u} f_j}$$

Short method $\bar{x}$ = $$A + \frac{\sum_{j=1}^{u} f_j d_j}{\sum_{j=1}^{u} f_j}$$

(8) Coding method $\bar{x}$ = $$A + \left\{\frac{\sum_{j=1}^{u} f_j u_j}{N}\right\} c$$

~~EE 401 (CRT analysis)~~

Standard deviation $$S = \sqrt{\frac{\sum_{j=1}^{u} f_j (x_j - \bar{x})^2}{N}}$$

(9)
$(\bar{x} - S) \to (\bar{x} + S)$
$(\bar{x} - 2S) \to (\bar{x} + 2S)$
$(\bar{x} - 3S) \to (\bar{x} + 3S)$

(10)
$(\bar{x} - S) \to (\bar{x} + S)$ [illegible] $\frac{\text{}}{\text{Total}}$ × 100
$(\bar{x} - 2S) \to (\bar{x} + 2S)$ ~~tot~~ Number / Total × 100
$(\bar{x} - 3S) \to (\bar{x} + 3S)$ Number × 100 [illegible]

(11) [illegible]

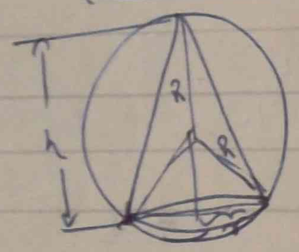
2nd derivative method

$\frac{dy}{dx} = 0$ ဖြစ်သော x အတွက် $\frac{d^2y}{dx^2}$ ရှာပါ။

+2 (min)

−2 (max)

(12) **max-and min: Problem**



$V = \frac{1}{3}\pi r^2 h$

$\frac{dV}{dr} = 0$ ဆိုကြပါစို့

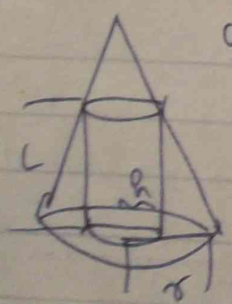
$\therefore V = \frac{1}{3}\pi r^2 h$

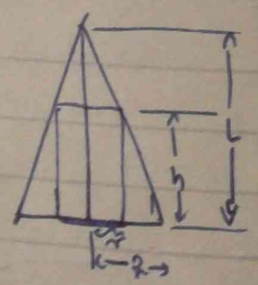
$h = R + \sqrt{R^2 - r^2}$

$\therefore V = \frac{1}{3}\pi r^2 [R + \sqrt{R^2 - r^2}]$

$V = \frac{1}{3}\pi r^2 R + \frac{1}{3}\pi r^2 \sqrt{R^2 - r^2}$

d





$V = \pi r^2$

$\frac{dV}{dr} = 0$

$\frac{h}{L} = \frac{L-h}{L} = \frac{r}{R}$

$L = RL - Rh = Lr$

$RL - Lr = Rh$

$L = Rh/R-r$

$V = \frac{L}{L-h} = \frac{R}{r}$

$Lr = RL - Rh$

$Rh = L(R-r) \quad \therefore h = \frac{L(R-r)}{R}$

$V = \pi r^2 \frac{L(R-r)}{R}$

$\frac{dV}{dr} = \frac{d\,\pi r^2 L}{dr} - \frac{d\,\pi r^2 \frac{Lr}{R}}{dr}$

$= \frac{d\,\pi r^2 L}{dr} - \frac{d\,\pi r^3 L/R}{dr}$

$0 = 2\pi r L - 3\pi r^2 L/R$

$\therefore 2\pi r L = 3\pi r^2 L/R$

$2 = \frac{3r}{R} \quad \therefore r = \frac{2}{3}R$

(13) **Integrals**

(1) $\int (x^2-x)^4 (2x-1)\,dx$

$u = (x^2-x)^4 \quad du = 4(x^2-x)^3 d(x^2-x)$

$= 4(x^2-x)^3 (2x-1)\,dx$

$\int (x^2-x)^4 (2x-1)\,dx \qquad (2x-1)\,dx = \frac{du}{4(x^2-x)^3}$

$\int u \{4u^{3/4}\}$

$4u^{7/4}\,dx$

$4(x^2-x)^{7/4}\,dx$

$u = (x^2-x)^4 = u^{1/4}$

$(x^2-x) = u$

$4(x^2-x)^3 = 4u^{3/4}$

(ii) $\int \frac{(x+1)\,dx}{\sqrt{x^2+2x-4}}$

$u = (x^2+2x-4)^{-1/2}$

$du = -\frac{1}{2}(x^2+2x-4)^{-3/2}(2x+2)\,dx$

$du = -(x^2+2x-4)^{-3/2}\,\underbrace{(x+1)\,dx}$

$(x+1)\,dx = \frac{du}{-(x^2+2x-4)^{-3/2}} = -\frac{1}{u^3}$

$\int (-\frac{1}{u^3}) \times du =$

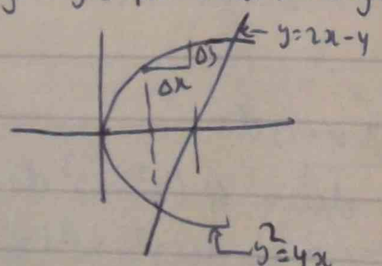
$\int_0^{2\pi} \sin^{1/2} t\,dt = \int_0^{2\pi} 2\sin^{1/2} t$

$= \frac{1}{2}\,2\int_0^{2\pi} \sin^{1/2} t\,dt = 2\int_0^{2\pi} \sin^{1/2} t\;\; d\,{}^1\!/_2 t$

$= 2\int_0^{\sim\pi} \sin t/2\; d\,t/2$

$= \left[2\cos t/2\right]_0$

fg $y^2 = 4x$ and line $y = 2x-4$



$y = 2x-4$

$\Delta y = \sqrt{4x} - (2x-4)$

$= 2\sqrt{x} - (2x-4)$

$= 2\sqrt{x} - 2x + 4$

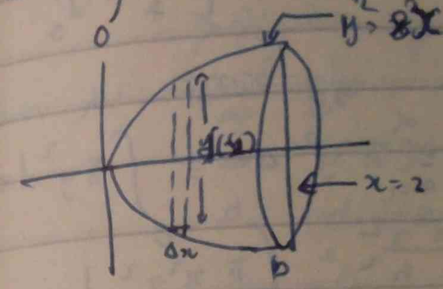
area: $\frac{1}{2}(2\sqrt{x} - 2x + 4)\,\Delta x$

$dA = (2\sqrt{x} - 2x + 4)\,\Delta x$

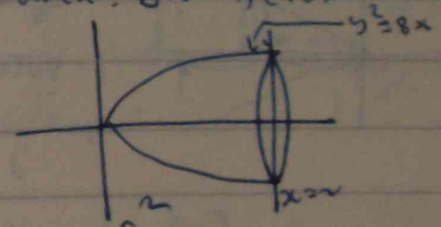
$\Delta x \cdot \Delta y = \Delta x \times 2\sqrt{x}$

$\therefore = 4\sqrt{x}\,\Delta x$

total $= \int_0^1 4\sqrt{x}\,\Delta x + \int_1^4 (2\sqrt{x} - 2x + 4)\,\Delta x$

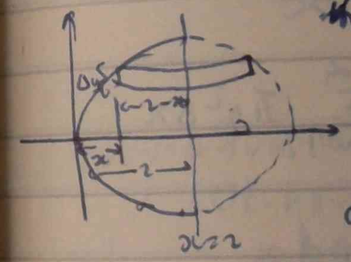


volume: $\int_a^b \pi y^2\,dx$ →



$\int_0^2 \pi \times (8x)\,\Delta x$

volume $= \sum_{n=1}^{\infty} \pi y^2\,\Delta x = \int_a^b \pi y^2\,dx = \int_a^b \pi 8x\,dx = 8\pi\left[\frac{x^2}{2}\right]_0^4 =$



$y^2 = 8x$

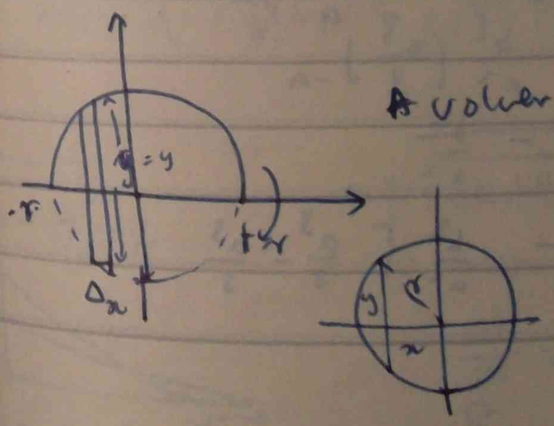
Volume: $\sum_{n=1}^{\infty} \pi(2-x)^2\,\Delta y$

$= \int_a^b \pi(2-x)^2\,dy = \int_a^b \pi\left[2 - \frac{y^2}{8}\right]^2 dy$

$a = y_1 = \sqrt{8\times 0} = 0$

$y_2 = \sqrt{8\times 2} = 4$

$= \int_0^4 \pi\left[4 - \frac{y^2}{2} + \frac{y^4}{64}\right] dy = \int_0^4 \pi 4\,dy - \int_0^4 \frac{\pi y^2}{2}\,dy + \int_0^4 \frac{\pi y^4}{64}\,dy$



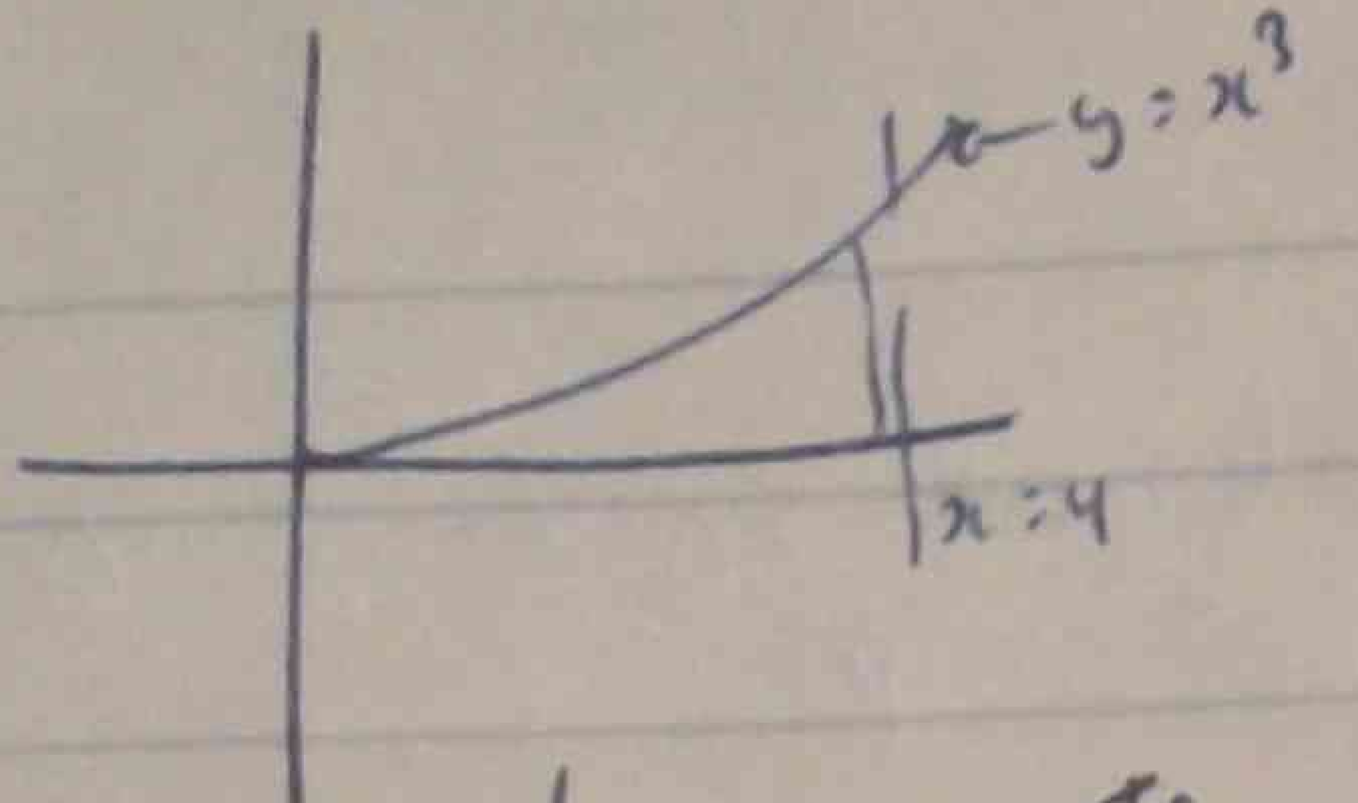
A volume: $\sum_{n=1}^{\infty} \pi y^2\,\Delta x = \int_{-r}^{r} \pi y^2\,dx$

$y^2 = \sqrt{r^2 - x^2}$

$\int_{-r}^{r} \pi(r^2 - x^2)\,dx = \int_{-r}^{r} \pi r^2\,dx - \int_{-r}^{r} \pi x^2\,dx$

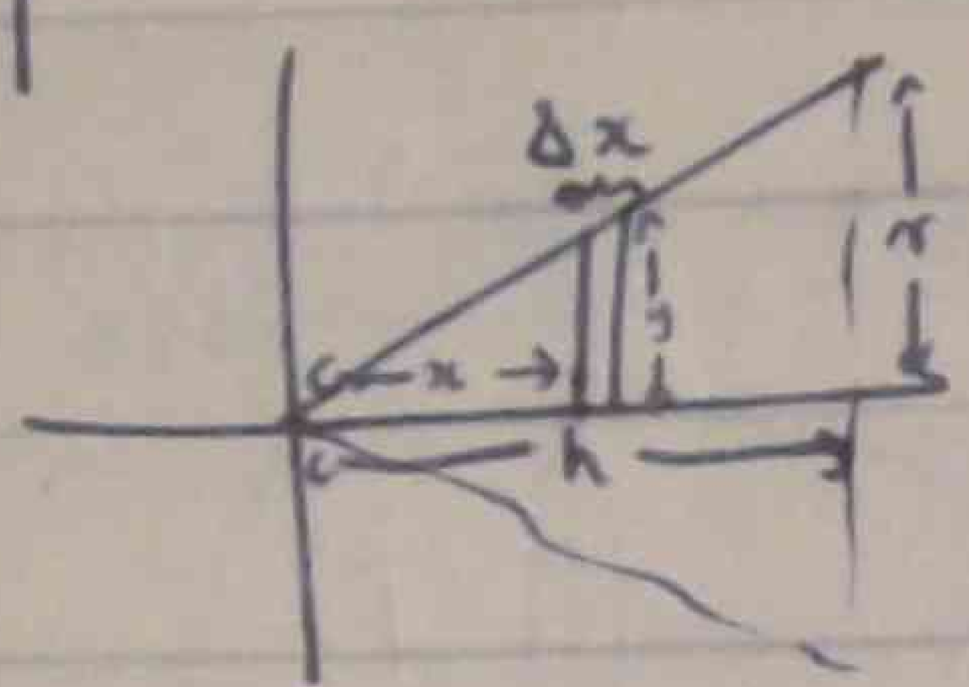
$\left[\pi r^2 x\right]_{-r}^{r} - \left[\frac{\pi x^3}{3}\right]_{-r}^{r}$

$2\pi r^3 - \frac{\pi r^3}{3} = 4/3\,\pi r^3$



$V = \int \pi y^2 \Delta x$

$= \int \pi x^6 dx \quad \left[\frac{x^7}{7} \pi \right]_0^4$



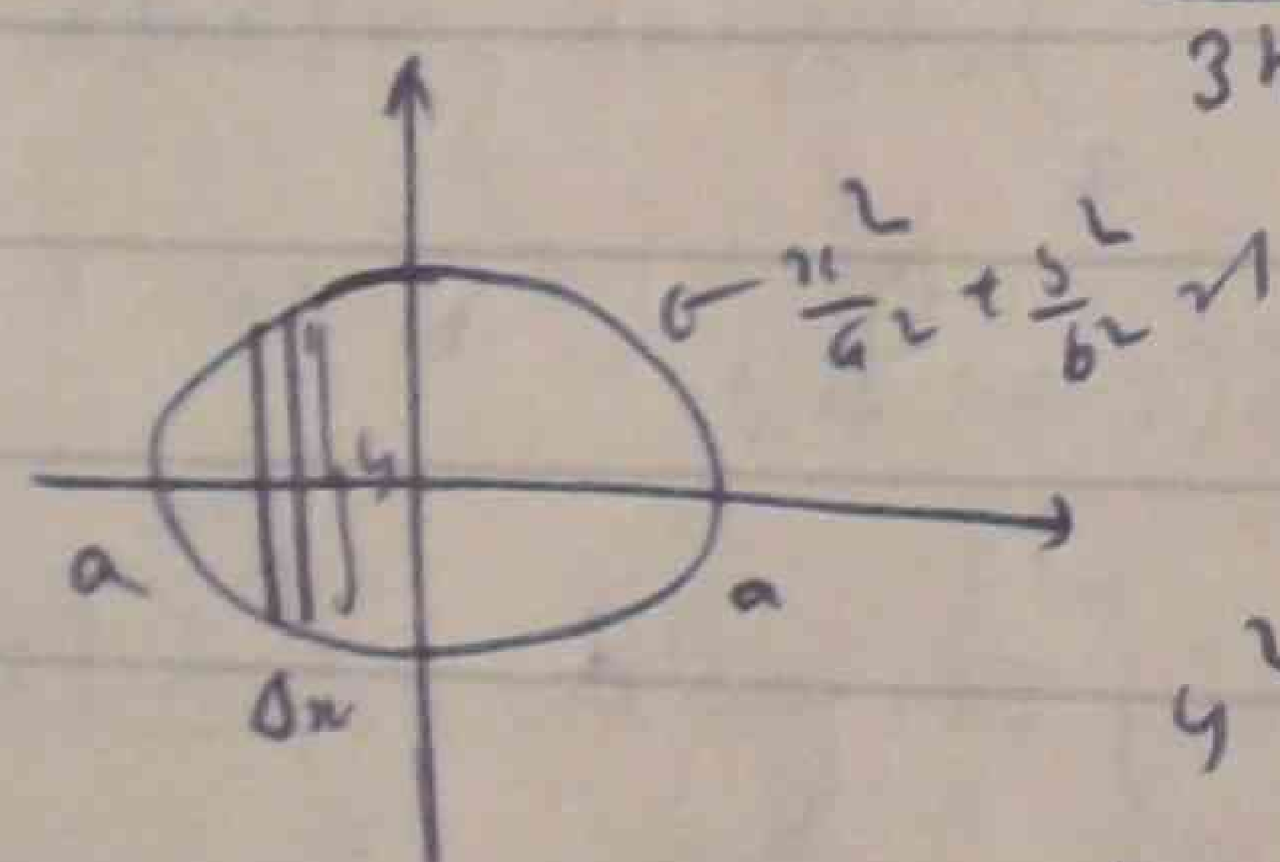
$\frac{y}{r} = \frac{x}{h} \qquad y = \frac{xr}{h}$

$\text{volume} = \sum_{m=1}^{\alpha} \pi y^2 \Delta x = \int_0^h \pi y^2 dx$

$y^2 = \frac{x^2 r^2}{h^2} = \int_0^h \pi \frac{x^2 r^2}{h^2} dx$

$\text{vol} = \left[\pi \frac{r^2}{h^2} \frac{x^3}{3} \right]_0^h$

$= \pi r^2 \frac{h^3}{3h^2} = \frac{\pi r^2 h}{3}$ #



$\text{volume} = \sum_{m=1}^{\alpha} \pi y^2 \Delta x$

$y^2 = b^2 \left[1 - \frac{x^2}{a^2} \right]$

$V = \int_{-a}^{a} \pi b^2 \left(1 - \frac{x^2}{a^2}\right) dx$

$V = \int_{-a}^{a} \pi \left(b^2 dx - \frac{b^2 x^2}{a^2} dx\right)$

$= b^2 [x]_{-a}^{a} - \frac{b^2}{a^2} \left[\frac{x^3}{3} \right]_{-a}^{a}$

$= - \left[\frac{b^2}{a^2} \frac{a^3}{3} - \frac{b^2}{a^2} \right]$ (crossed out)

$= b^2 [a + a] \pi - \frac{b^2}{a^2} \left[\frac{a^3}{3} - \frac{a^3}{3} \right]$

$= 2\pi a b^2$ #

Differential

maxima, minima

① [illegible] equation [illegible] constant [illegible] variable [illegible]

eg. $V = \pi r^2 h$

var (r) const. (h) ← const : [illegible] differentiate [illegible]

② A [illegible] A [illegible]

[illegible] V, A → r ∴ r → A → V

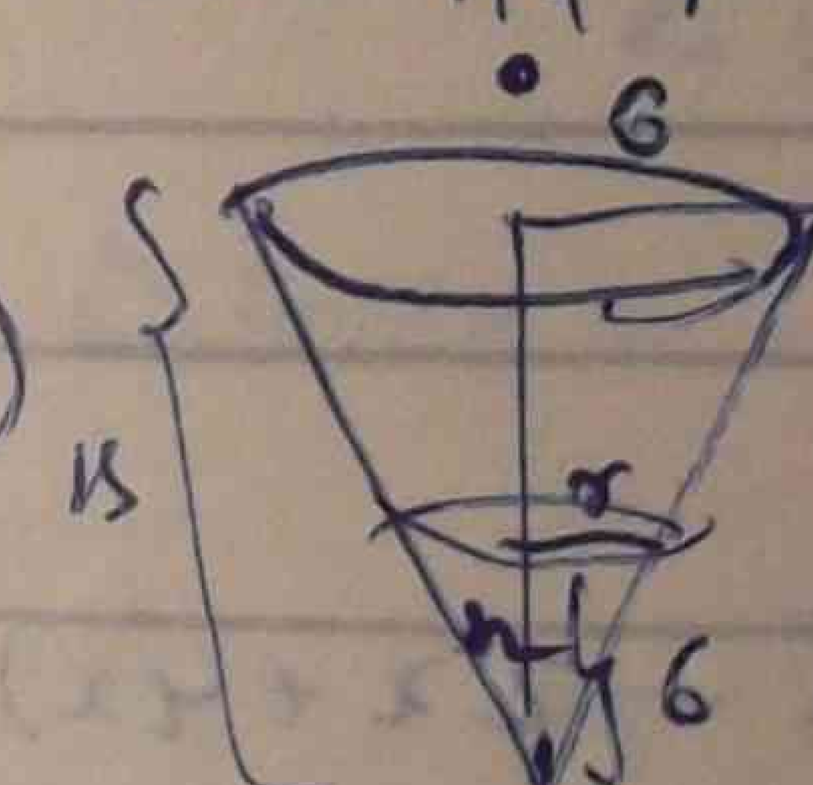
[illegible]

$\therefore \frac{dV}{dA} = \frac{dV/dt}{dA/dt}$

[illegible]

(or) $\frac{dV}{dr} = \frac{dV/dt}{dr/dt}$

③



$\frac{dV}{dt} = 6\,m^3/\text{min}$

$\frac{dh}{dt} = ?$

$V = \pi r^2 h \qquad h = 6 \text{ [illegible]} \quad \frac{dh}{dt} = ?$

var. (r) var. (h) [illegible]

$\frac{r}{6} = \frac{6}{15}$

$r = \frac{36}{15} = \frac{12}{5}$

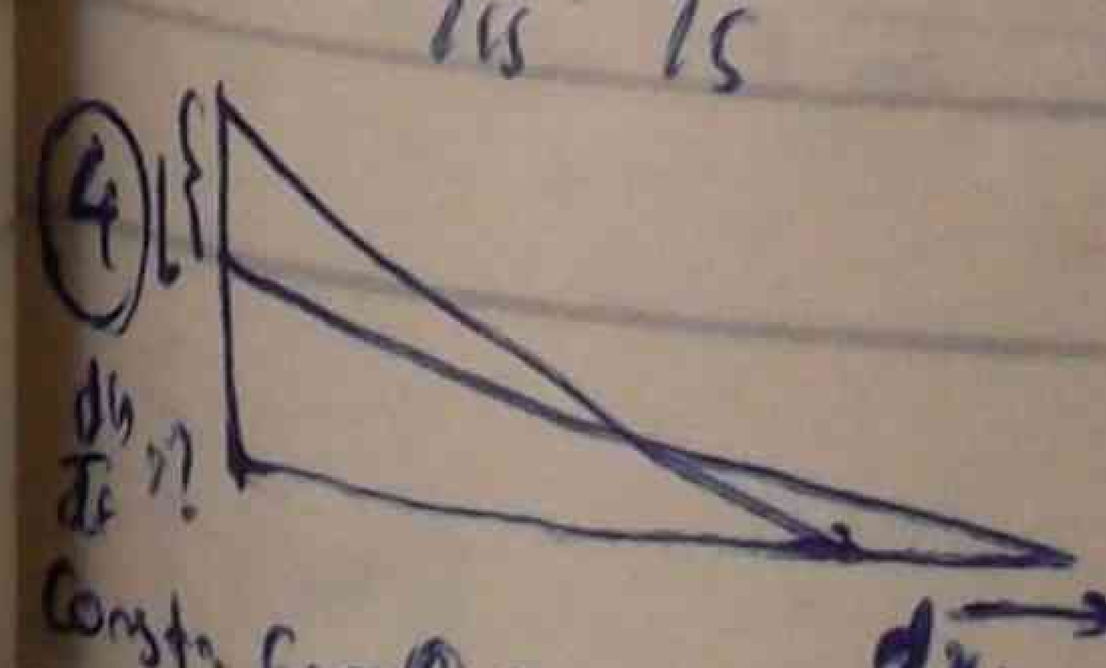
[illegible] var [illegible] $\frac{dh}{dt}$ [illegible] var [illegible]

$\therefore V = \pi \left(\frac{12}{5}\right)^2 h$

$\frac{dV}{dt} = \frac{\pi 144}{25} \frac{dh}{dt}$

[illegible]

④



Const. [illegible] eq. [illegible] diff [illegible]

(5) ဥပမာ။

Δ area: $\Delta x \to dx$ / dx

$$\triangle ABCD = \int_1^4 (2\sqrt{x} - 2x + 4x)\,dx \quad \{1 မှ 4 အထိ$$

$y_1^2 = 4x \quad \therefore y_1 = 2\sqrt{x}$

$\Delta y = y_1 - 0 = 2\sqrt{x}$

$ACOE: \quad \Delta x \times \Delta y_1$

$= \Delta x \times 2\sqrt{x}$

$= 2\sqrt{x}\,\Delta x$

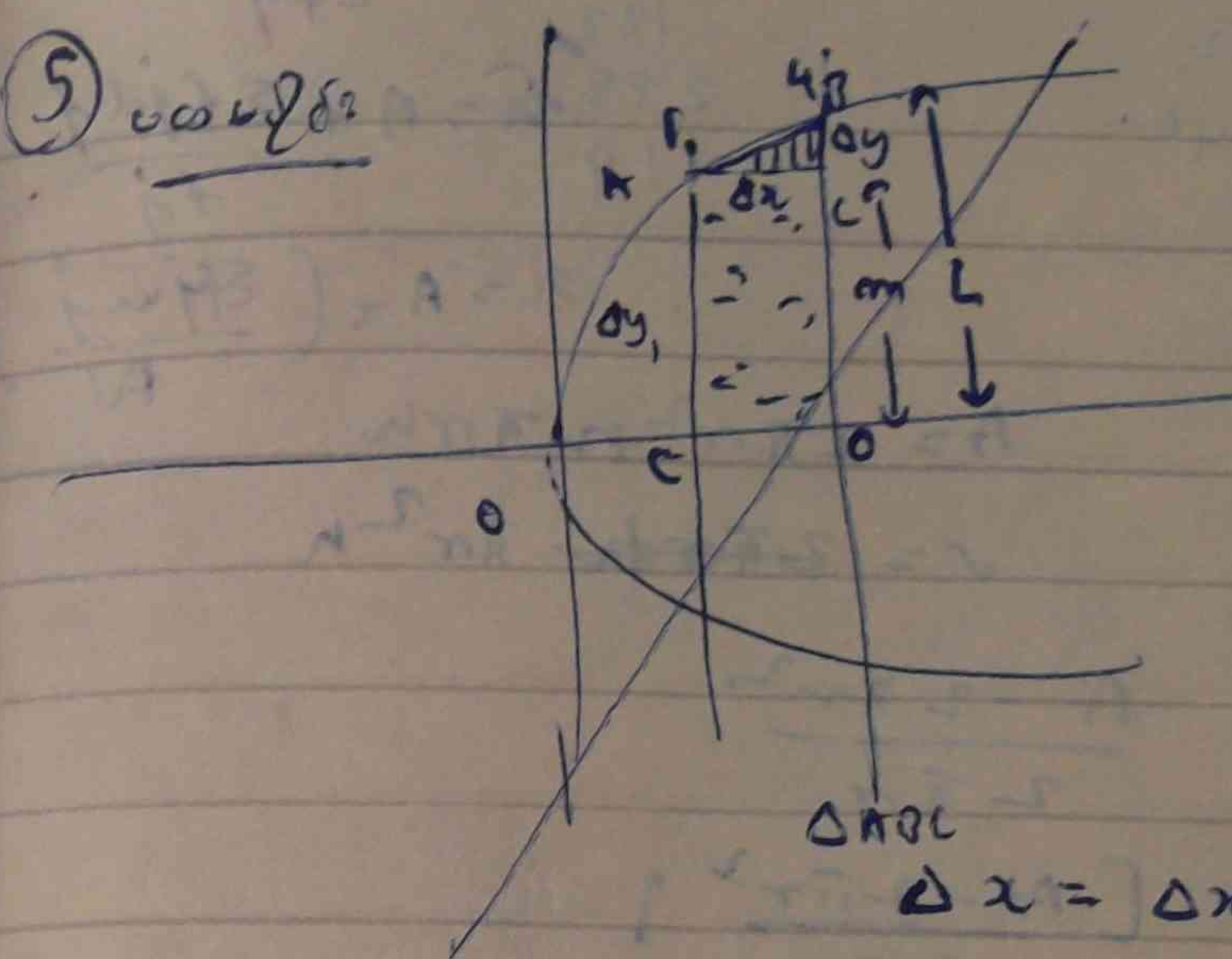
$2\,\square ACOE: \quad 4\sqrt{x}\,\Delta x$

$\Delta x \to dx \quad dx$

$= 4\sqrt{x}\,dx \qquad \{0 မှ 1 အထိ \quad 2\,\square ACOE = \int_0^1 4\sqrt{x}\,dx$

$$\text{Total } ABCDE = \int_0^1 4\sqrt{x}\,dx + \int_1^4 (2\sqrt{x} - 2x + 4x)\,dx$$

(5) ပထမနည်း



$\triangle ABC$

$\Delta x = \Delta x$

$\Delta y = L - m$

$=$ curve ရဲ့ y တန်ဖိုး $-$ line ရဲ့ y တန်ဖိုး

$= 2\sqrt{x} - (2x - 4)$

$= 2\sqrt{x} - 2x + 4$

$\Delta x = \Delta x$

Δ area $= \frac{1}{2}\Delta y \times \Delta x \quad$ ($\square$) $\Delta y \times \Delta x$

$\triangle \therefore$ area $= (2\sqrt{x} - 2x + 4)\,\Delta x$

$\Delta x \to dx \quad \therefore$ ~~$\lim_{\Delta x \to 0}$ (area $2\sqrt{x}\,\Delta x$~~

~~$\Delta x \to 0 \therefore \Delta x = dx$ area $= \int_0^1 2\sqrt{x}\,dx$~~ $- \int_1^4 (2x-4)\,dx$

curve   line

$\frac{\Delta x \times \Delta y \times 2}{\Delta x \times 2\sqrt{x} \times 2}$

$y^2 = 4x \,/\, y = 2\sqrt{x}$

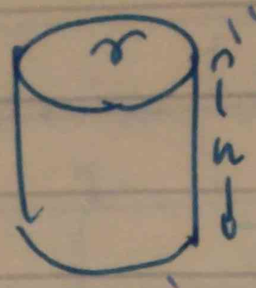
$y = 2x - 4$

$\leftarrow$ ဒုတိယနည်း

$$\bar{x} = \frac{\Sigma f_j x_j}{\Sigma f_j}$$

$$\bar{x} = A + \frac{\Sigma f_j d_j}{f_j}$$

$$\bar{x} = A + \left(\frac{\Sigma f_j u_j}{N}\right)$$



$$A = 2\pi r^2 + 2\pi rh$$

$$V = \cancel{2\pi rh}\ \pi r^2 h$$

$$h = \frac{A - 2\pi r^2}{2\pi r}$$

$$V = \pi r^2 \left[\frac{A - 2\pi r^2}{2\pi r}\right]$$

$$= \frac{Ar}{2} - \pi r^3$$

$$0 = \frac{Ar}{2} - \pi r^3$$

$$Ar = 2\pi r^3$$

$$A = 2\pi r^2$$

$$r = \sqrt{\frac{A}{2\pi}}$$

$$V' = A/2 - 3\pi r^2$$

$$0 = A - 6\pi r^2$$

$$r = \sqrt{\frac{A}{6\pi}}$$

$$h = \frac{\sqrt{A/6\pi} - 2\pi \sqrt{A/6\pi}}{2\pi \sqrt{A/6\pi}}$$

$$x^2 + y^2 = r^2$$

$$y^2 = r^2 - x^2$$

$$\frac{dy^2}{dx} = \frac{dr^2}{dx} - \frac{dx^2}{dx}$$

$$2y\frac{dy}{dx} = 2r\frac{dr}{dx} - 2x$$

ရက်စွဲ: ၁၀ ကနှစ် ၂၀၁၇ ဧပြီ

ခေါင်းစဉ်: ၇။ စစ်ဆေးခြင်း

၁. အလုပ်မှ ရှုသည့် အလုပ်ပြီး မှတ်တမ်း ပြုလုပ် ရမည့် လုပ်ငန်း ကာလ မှတ်

၂. ကအလုပ် ပါဝင်သူ ဆော်ဒေး ... ကြည့်ရမည်

၃။ control Room ကိုစစ်ဆေးရမည်

(1) ~~Eng~~ ထိန်းချုပ်ခုံကို စစ်ဆေးခြင်း

(2) Engine speed ကိုစစ်ဆေးခြင်း

(3) နှင်း speed နှင့်ဆိုင်ရာ pressure gauge များ မှန်မမှန် အားလုံး ပြမပြ စစ်ဆေးရမည်။

(4) cylinder တစ်ခုချင်းစီ၏ Exhaust temperature ကိုစစ်ဆေးရမည်။

မှတ်ချက် – temperature အများဆုံးနှင့် အနည်းဆုံး ကိုသတိထားရမည်။

(5) main Engine ထိန်းချုပ်မှုကို Bridge (သို့) Engine room အလယ်ဗဟိုမှ သာသည်ကို ကြည့်ရမည်

(6) main switch Board ကိုစစ်ဆေးခြင်း

(7) အလုပ် generator မောင်းထားသည် ကိုကြည့်ရမည်

(8) မောင်းထားသော generator ၏ load (kw), ampere voltage, Frequency တို့ကိုစစ်ဆေးရမည်။

(9) Steering power အလုပ်လုပ်ခြင်း ရှိမရှိ ကိုကြည့်ရမည်

2nd term

10 ... unuseful ... Power ...

2 ways

① { ... galley Power ...
  ... galley Power ...

② { winch Power ... Bridge ...
  ... (... Bridge ...)

~~3) Boiler~~

11 Engine ... Diesel ... ①, ②, ③ ...

(I) Generator Engine ...

) generator Engine ...

✓ (1) lubricating pressure
✓ (2) Jacket pressure
(3) Expansion tank water level
✓ (4) lubrication gauge ...
Jacket Pressure gauge ... level ...

(5) lubrication oil pressure gauge ..., Jacked temperature ...
... lubrication oil sump level ...
... Expansion / radiator tank / Freshwater level ... Normal ... adjusting ...

✓ (6) sea water cooling system ... attachment ... Belt ...

✓ (7) generator ... Floor plate ...

(8) ~~...~~ Double bottom tank ... clean ...

✓ generator ... ① ... ② ... ③ ... ④ ...

II Main Engine ကိုစစ်ဆေးခြင်း

Main Engine တွင် (1) lubricating system
(2) cooling system ကို ကိုစစ်ဆေးရမည်

lubricating system ကိုစစ်ဆေးခြင်း

(1) lubrication swamp level ကိုကြည့်ခြင်း
(2) Turbo Charger Turbo oil level ကိုကြည့်ခြင်း
ကိုပိုင်းခြား (1) compressor side
(2) Turbine side [illegible]
ဆေးခြင်း
(3) Raker arm oil level (lubrication oil)
ကိုစစ်ဆေးခြင်း
(4) gear oil ~~ကို~~ swamp level ကိုစစ်ဆေးခြင်း

III Bearing စစ်ဆေးခြင်း

① Tunnel Bearing
Propeller bearing lubrication oil level
ကိုစစ်ဆေးခြင်း

② Tunnel bearing temperature gauge ကိုကြည့်ခြင်း

③ Tunnel bearing temperature gauge ကိုကြည့်ခြင်း

IV Gear ကိုစစ်ဆေးခြင်း

- gear oil pressure Normal ဖြစ်မဖြစ် (ahead/astern
[illegible] စစ်ဆေးခြင်း
- gear oil temperature ကို [illegible] စစ်ဆေးခြင်း
gear box ကို Checking ပြုခြင်း

V

cooler ကိုစစ်ဆေးခြင်း

gear oil ~~cooller~~ cooler
seawater temperature gauge [illegible]
Pressure | temperature စစ်ဆေးခြင်း

- main seawater pump pipe ကို စစ်ဆေးရမည်။
- Floor plate အောက်ရှိ ဆီ / ရေ [illegible] စစ်ဆေးရမည်
  [illegible] ရမည်။

(VII) Tank ကို စစ်ဆေးရမည်။

- Day tank / service tank oil level ကို စစ်ဆေးရမည်။
- drain ~~ကို စစ်ဆေးရမည်~~ ဖောက်ချရမည်။
- [illegible] ကို စစ်ဆေးရမည်။
- Expansion tank ရှိ water level ကို ကြည့်ရမည်။
  ရေလျော့လျှင် [illegible] ဖြည့်ရမည်။

[illegible] Engine room [illegible]
[illegible] DO / LO / water [illegible]
[illegible]

Date 26-12-83
RS
ME 28 DO 105
P NIL / S NIL LO $1\frac{1}{2}$ Drum

W F A
$\frac{1}{2}$ $\frac{1}{2}$

Engine speed 800 rpm (Full run away)

[illegible] pressure / temperature / current amperage

Pressure

main starting air - 6.4 bar → ahead - 6 bar, astern - 4 bar
lubricating oil after filter - 4 bar
Lubrication oil ~~aft~~ before Filter 4.4 bar
lub. oil Rocker arm 0.9 bar
gear oil 2.8 bar
~~Ahead~~ Engine speed - 4.2 bar
Charge air - 0.8 bar
Fuel oil 1.2 bar
Starting air 5 bar

clutch oil 20.5 bar
Freshwater 2 bar
seawater 2.3 bar

Alternater ②

A	A	A
50amp	50amp	50amp
5kw	60Ht	440V

alternater ③

A	A	A
60amp	65amp	65amp
kw	HZ	V
30kw	60Ht	440V

Steering gear ②

Amp — 5.5amp

အပိုင်း ၂။ Duty log ဖြည့်သွင်းရန် အချက် လေ့ကျင့်ခြင်း

Duty log ဖြည့်သွင်းရာ၌ ပါဝင်သော အချက်များ ရှိပြီး ဖြည့်သွင်းရန်

① watch counter
② RPM by counter, RPM by technometer
③ main fuel lever
④ governor adjustment
⑤ Engine room temperature
⑥ Seawater temperature
(i) before cooler
(ii) Sea water cooling after coolers —
- ① Freshwater
- ② Lubrication oil
- ③ Fuel | NO 3510
- ④ Bilage oil

⑦ Fre water cooling temperature
- (i) fresh water cooler in
- (ii) freshwater cooler out
- (iii) inlet Jacket
- (iv) cylinder discharge

1/2 3/4 5/6 7/8 9/10 11/12

(8) Pistan cooler in / out Temperatue

(9) lubrication oil cooler in / out Temperatue

(10) inlet piston temperatue

(11) Pistan cooling discharge
1/2 3/4 5/6 7/8 9/10 11/12 temperatue

(12) Nozzle cooling in/out temperatue

(13) Exhaust gas cylinder out let temperatue
1/2 3/4 5/6 7/8 9/10 11/12

(14) Heavy oil settling tanke 1/2 temperatue

(15) Heavy oil day tanke 1/2 temperatue

(16) Heavy oil heater temperatue

(17) Heavy oil end pump temperatue

(18) viscosity meter

(19) flow meter Fresh water generator plant pressure & temperature (i) Flowmeter

(20) (i) Vacum
(ii) salinity PPm
(iii) Ejetor pressure
(iv) condenser in let / out let
(v) evaporator in / out
(vi) evaporator in / out

(21) working hours

(22) provision Refrigerated set (i) gas scavenging, discharge & Lubrication oil pressure
(ii) sea water in / out temperatue

(23) Air conditioning plant
(i) gas pressure discharge / scavenging / pressure
(ii) Air temperatue
(iii) lubrication oil temperatue / pressure / level

(24) Boiler water

(25) generator of (i) KW (iii) frequency
(ii) r pm

(25) cooler temperature
(i) luboil in/at
(ii) seawater in/at
(25) Jacket temperature highest
lowest
(26) Exhaust temperature { highest / lowest

(27) seperators or Heavy oil
diesel oil } temp/Pressure
luboil

(28) indicator card

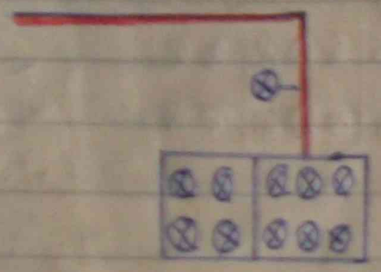
(29) Running hrs
(i) ~~gear~~ main Engine
(ii) generators
(iii) Boilers
(iv) compressors
(v) freshwater generator
(vi) Provision compressor
(vii) Air conditioner
(viii) cargo fridg:
(ix) energy fine pump

(30) Back pressure
(31) Turbo blower rpm upper / exhaust temp:
out let turbo
(32) scavenging air or temperatures { (i) ~~Blank~~ Blower inlet / (ii) out let / (iii) After cylinder
(33) reservoir pressure, temperature
(34) Turbo oil L Pressure temperature
(35) seawater temp
(36) sounding

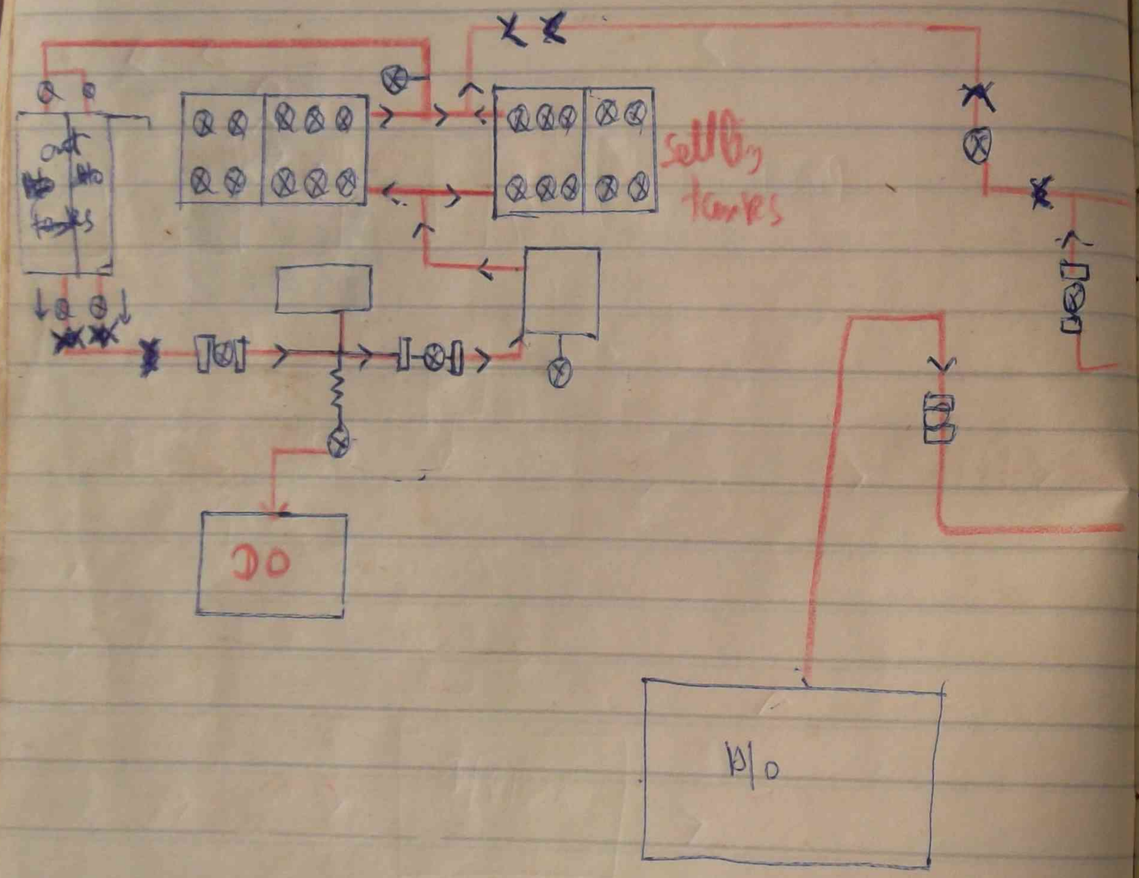
ঐচ্ছিক: Noon report সংগৃহীত
(i) total rev:
(ii) steaming time
(iii) distance by engine
(iv) " observer
(v) apparent slip
(vi) Detection

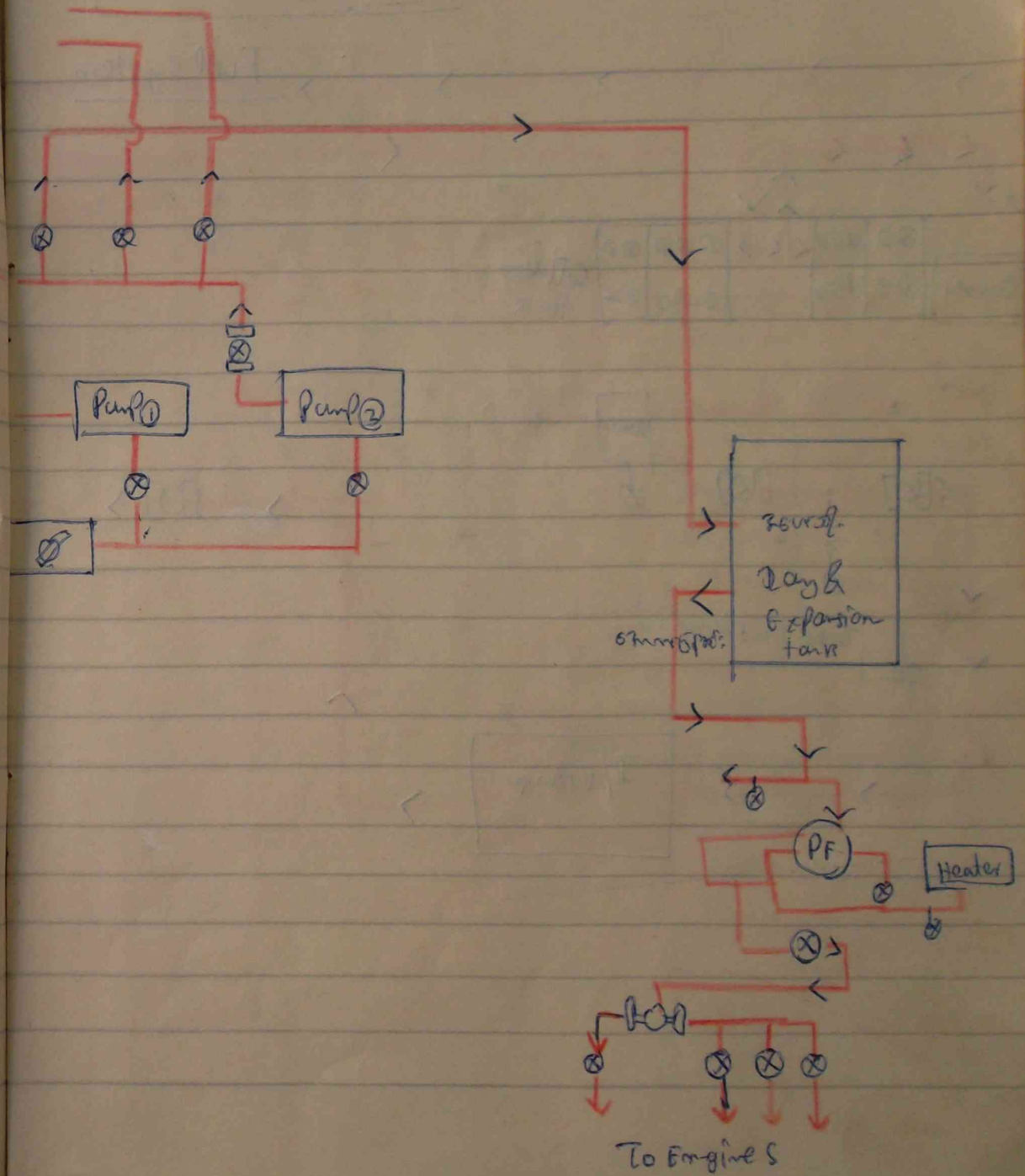
Financing

Fuel Oil System



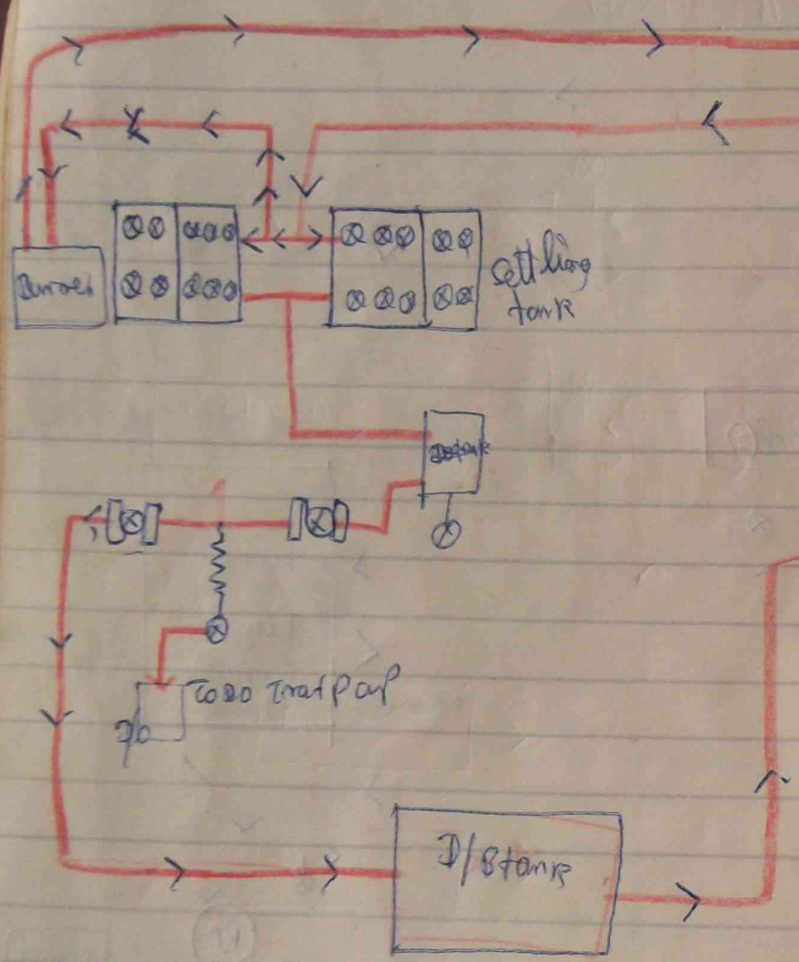
I Fuel oil system

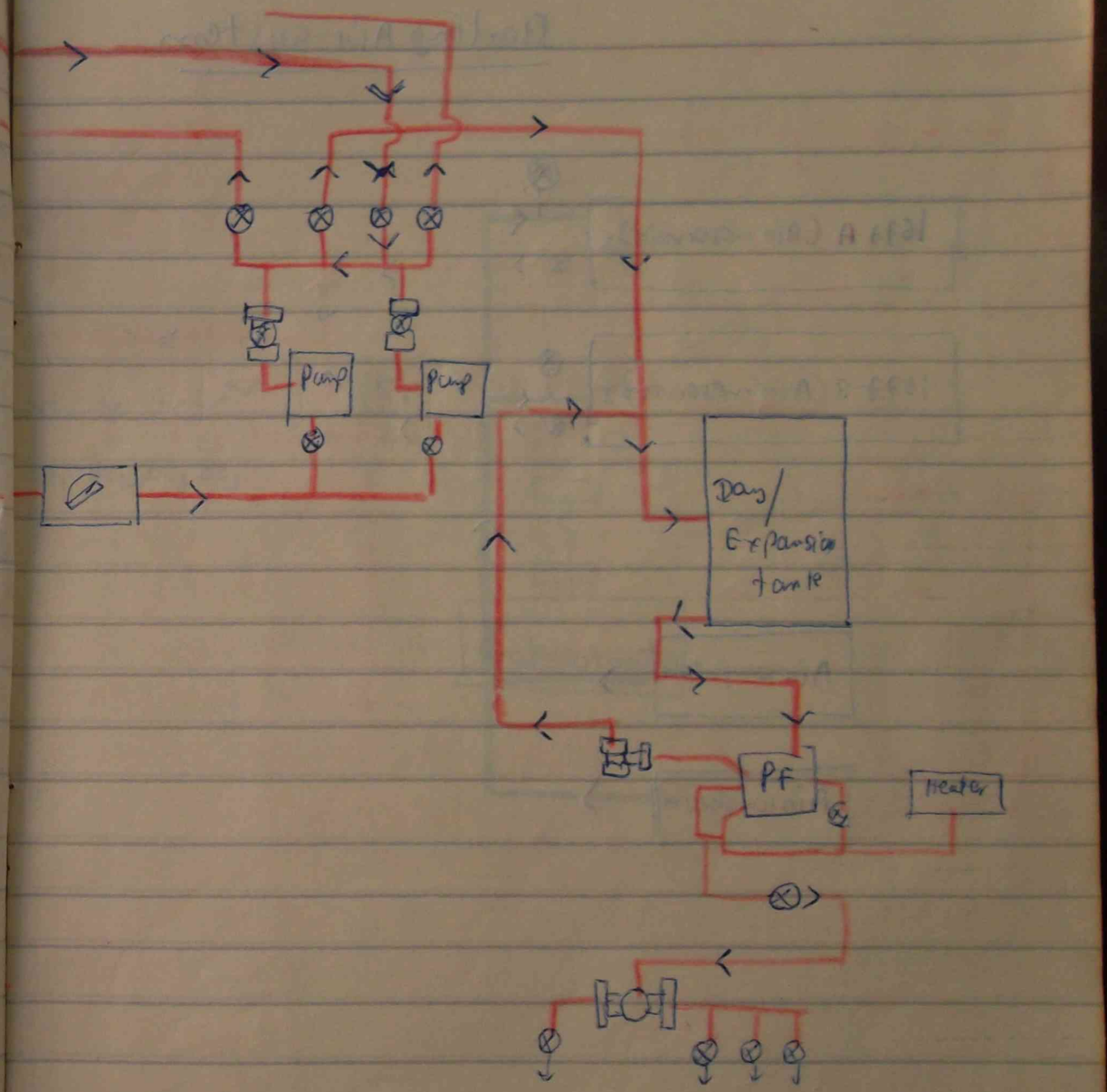




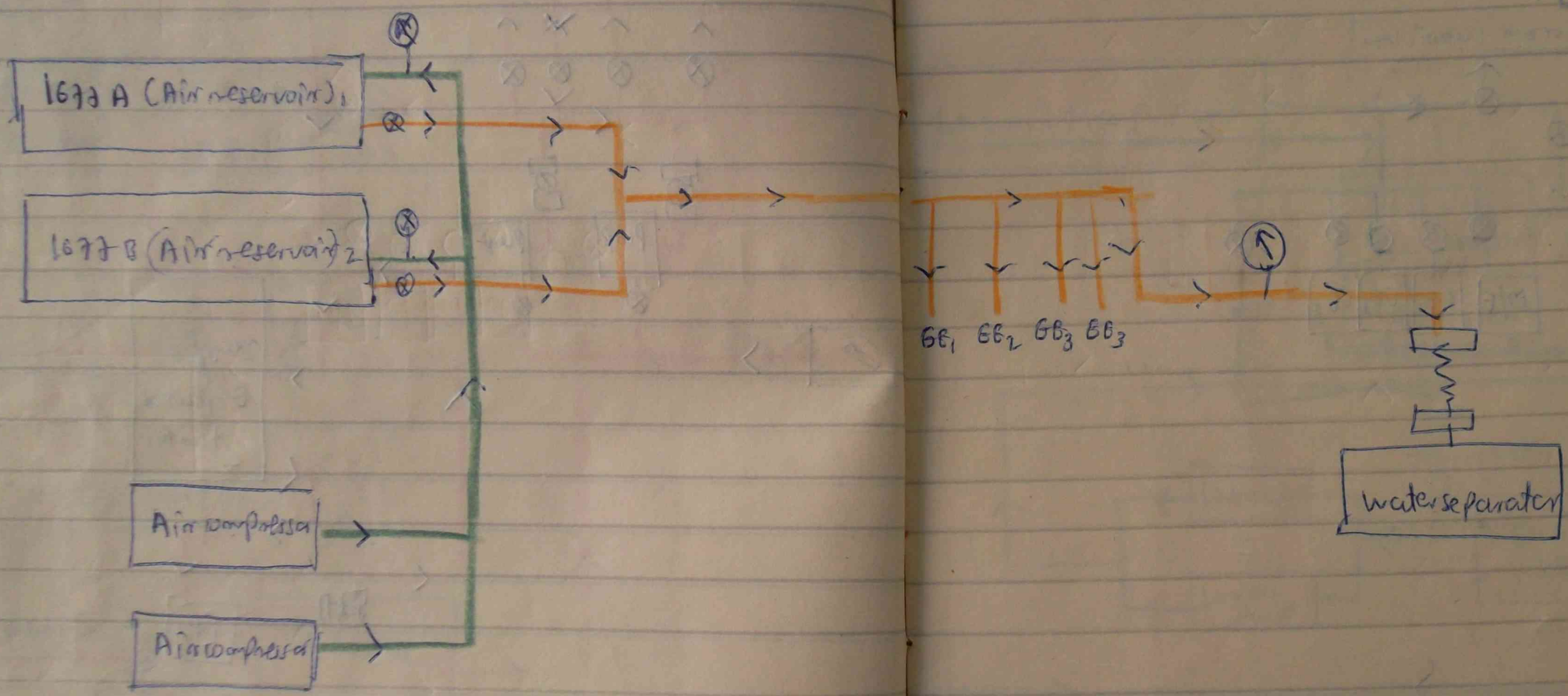
2nd (?) Pipe Tracing

Fuel system

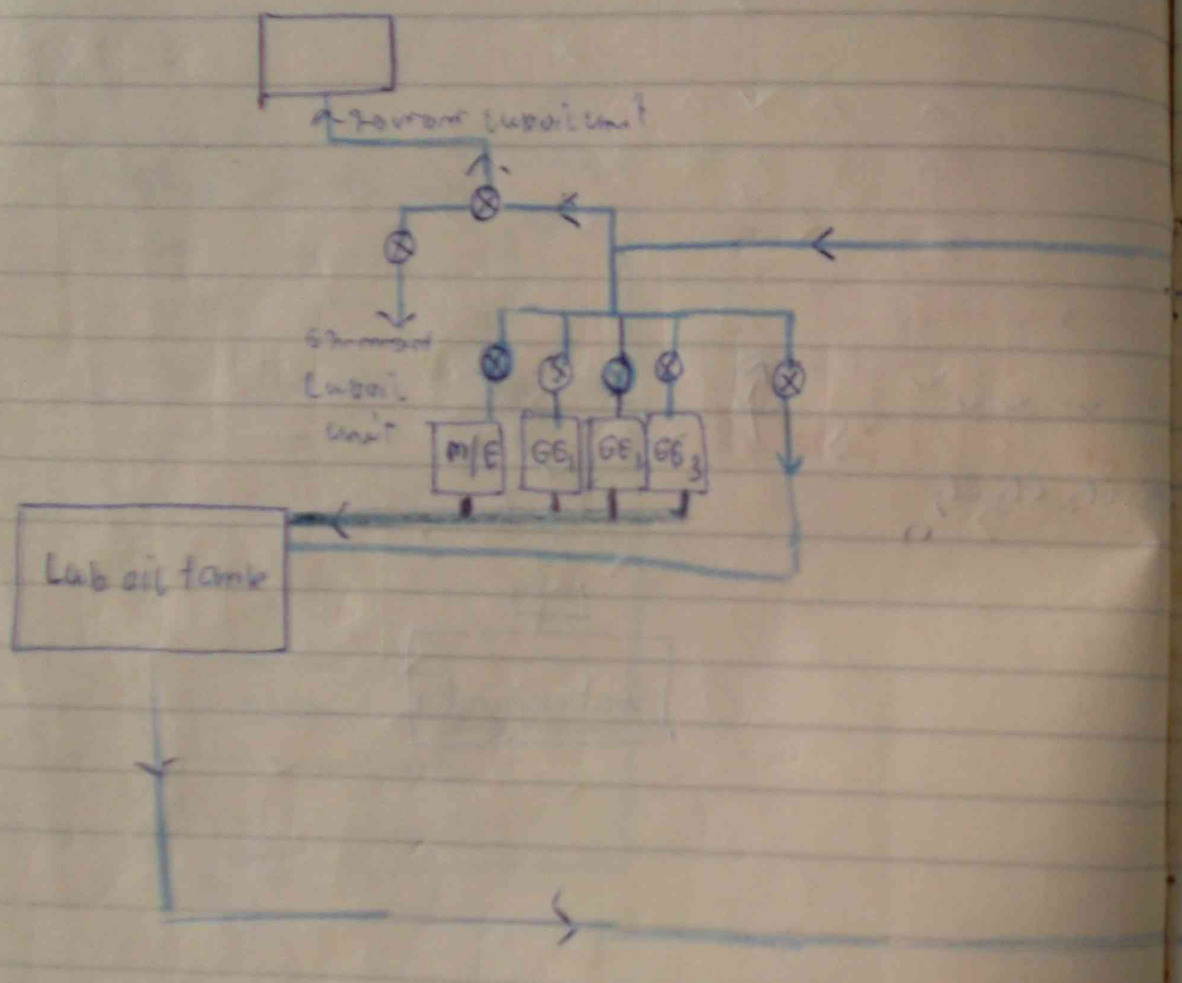


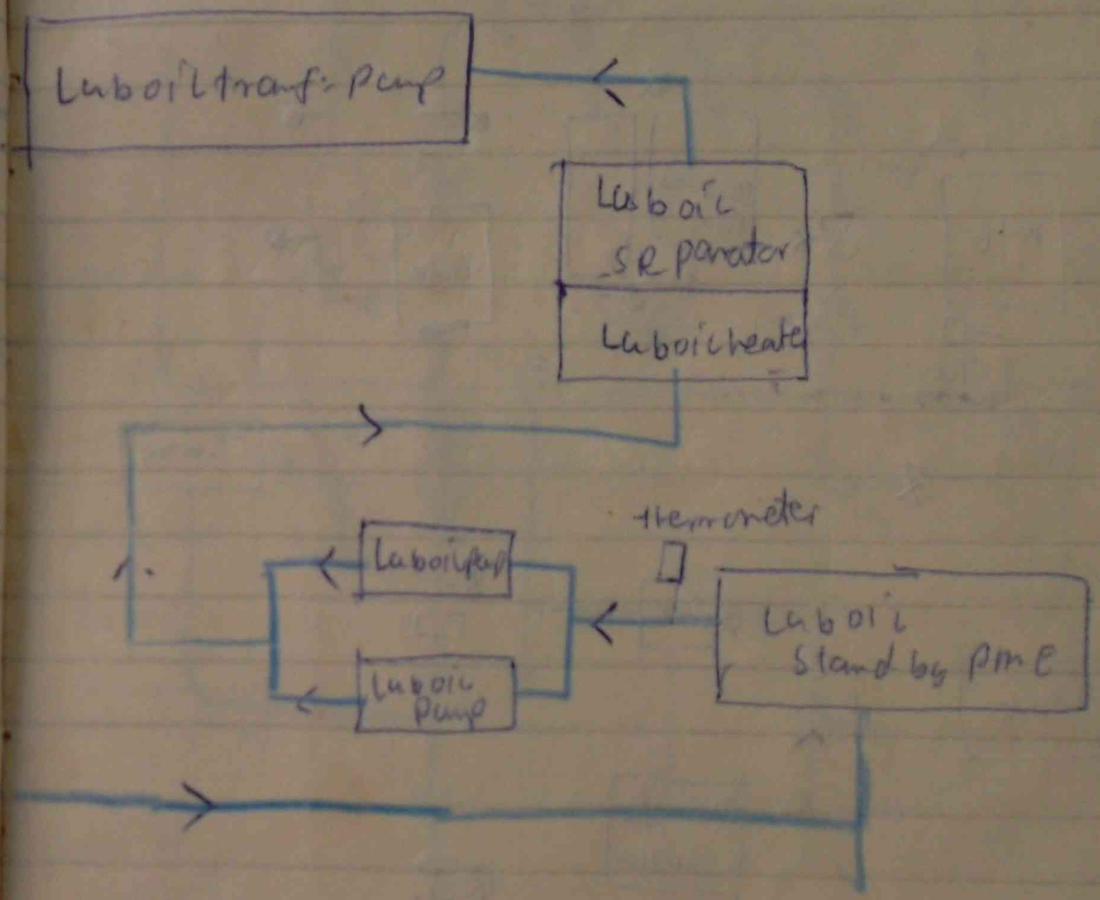


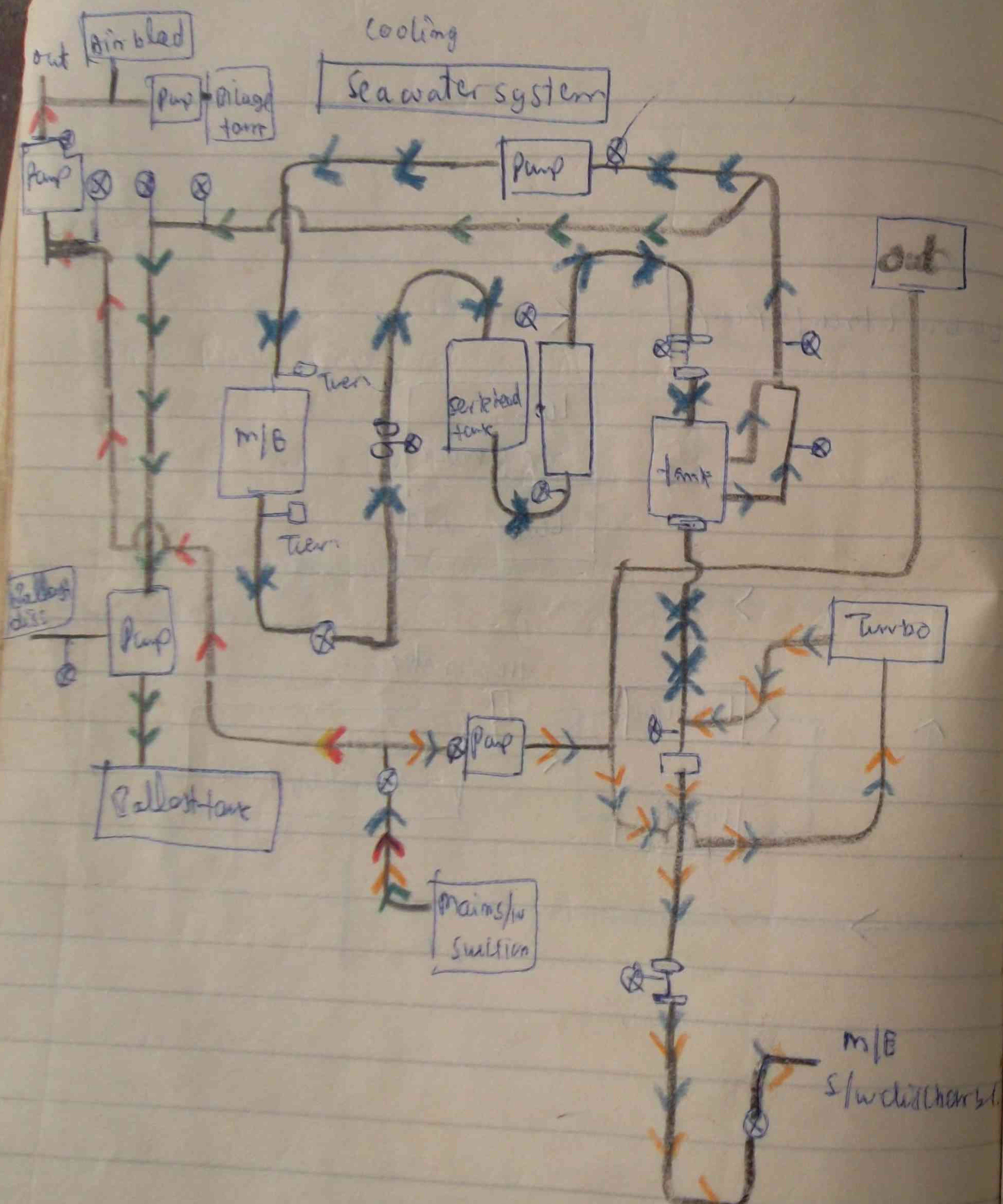
Starting Air system

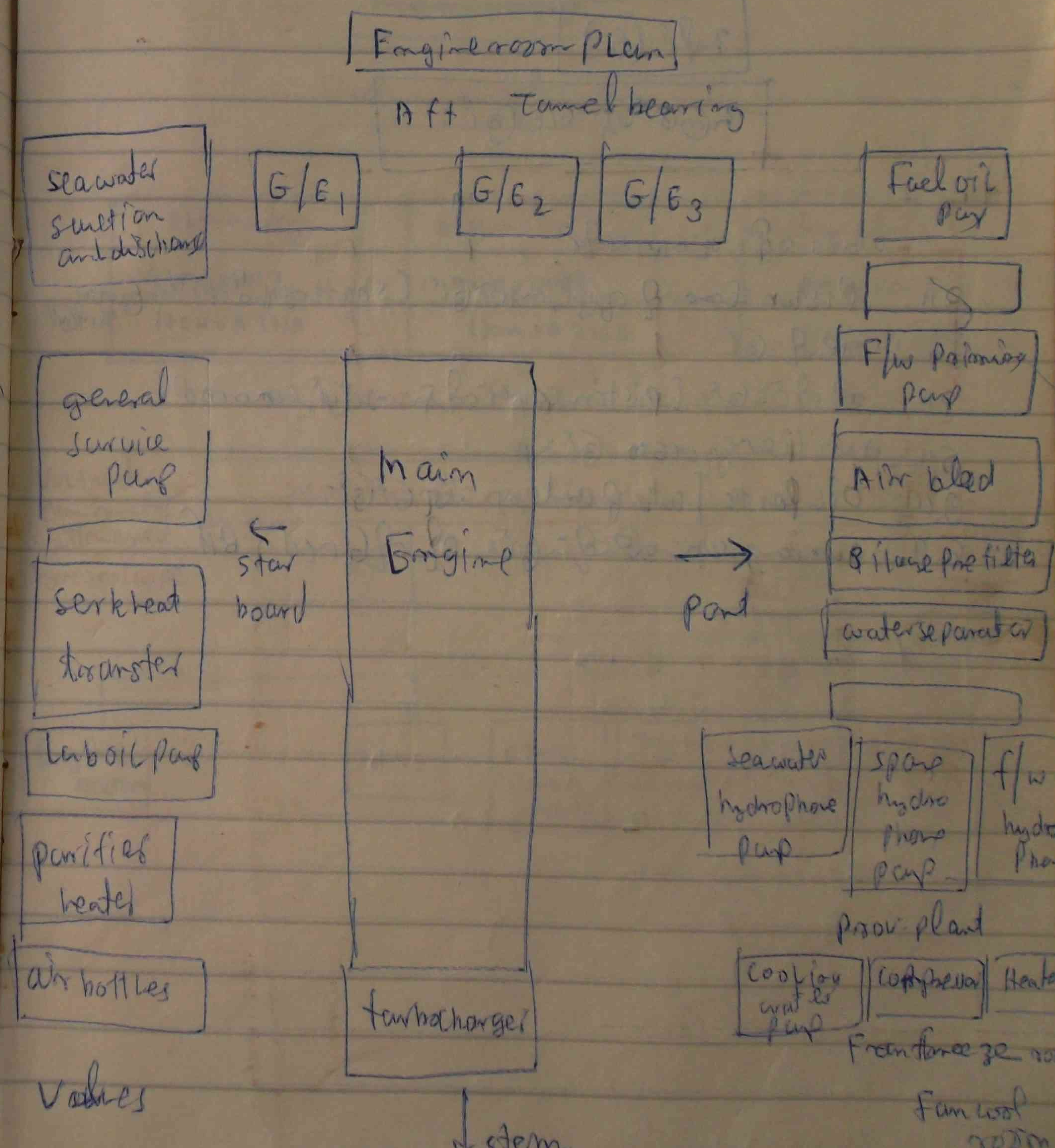


Lubrication System






Cooling
Seawater system
Air bled
out
Pump
Bilage tank
Pump
out
M/E
Turn
Turn
Serktead tank
tank
Turbo
Pump
Pump
Ballast tank
Mains/w suction
M/E
S/W discharge


Engine room Plan
Aft
Tunnel bearing
Seawater suction and discharge
G/E1
G/E2
G/E3
Fuel oil Pup
F/w Pairing pump
Air bled
Silone fine filter
water separator
general service pump
Star board
Main Engine
Port
Serkteat transfer
Lubeoil pump
purifier heater
Air bottles
turbocharger
Seawater hydrophore pump
spare hydro phore pump
f/w hydro phore
Prov plant
Cooling water pump
Compressor
Heater
Valves
stern

အပိုင်း (၄)

[illegible]

[illegible]

၁။ Piston ~~[illegible]~~ [illegible] (Shaft [illegible] Piston [illegible])

၂။ [illegible]

၃။ [illegible] (piston cover [illegible])

၄။ purifier [illegible]

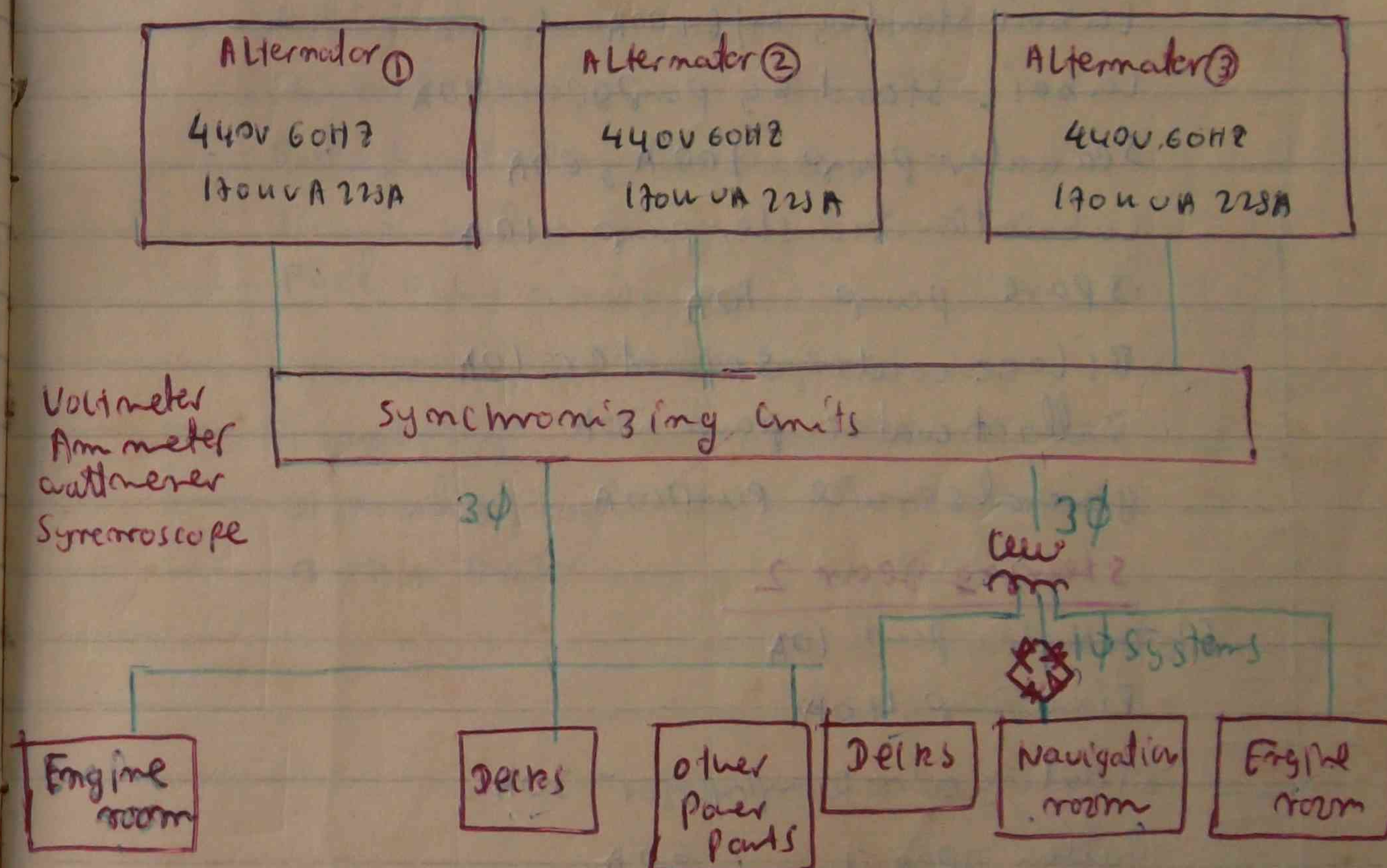
၅။ Oil Ports / lub ports [illegible]

၆။ bearing [illegible]

အပိုင်း (၅)

~~[illegible]~~ :

Electrical supply system



Engineroom 3φ power units

general service pump

Steering gear 1 (30A)

Lub oil standby m/E 100A
Lub oil stand by pump gear 100A
Seawater pump 100A, 60A
Lubrication transfer pump 10A
Spare pump 10A
Bilage water separator 10A
Ballast water pump 30A
general service pump 60A

Steering gear 2

F/O transfer pump 10A
Bilge pump 40A
Starting air compressor 30A
Jelling gear 2 60A
F/W pump
S/W pump
Lub oil pump
Lub oil heater
Purifier 2.6A 440V
Lub oil transfer pump 2.75HP 440V

Air compressor 1 440V 11HP
Air compressor 2 440V 11HP
Cooling water pump
Fan cool room
Ballast water pump 11HP 440V
Bilage ballast pump
Fuel oil priming pump
Bilage separator Autoplant
Bilage pump
Hydraulic pump
Anchor motor

Engine room 1φ power units

- Fan breeze room 3x10A
- Fan cool room 3x10A
- S/w seawater hydrophone 220v 1-3HP 60A
- Spare hydrophone pump
- Freshwater hydrophone pump
- galley distribution 100A
- Stantinany air room 20A
- Freshwater hydrophone 10A
- Freshwater cooling pump 60A
- provision plant 20A
- welding out let 30A
- Heater all vet - 100A
- Fan congo hold
- F/o Trf: pump
- Engine room Fans 30A, 30A
- galley fan 30A
- Emergency S/w board 100A
- Lubrication oil separater 10A
- Fan accomodation room 20A

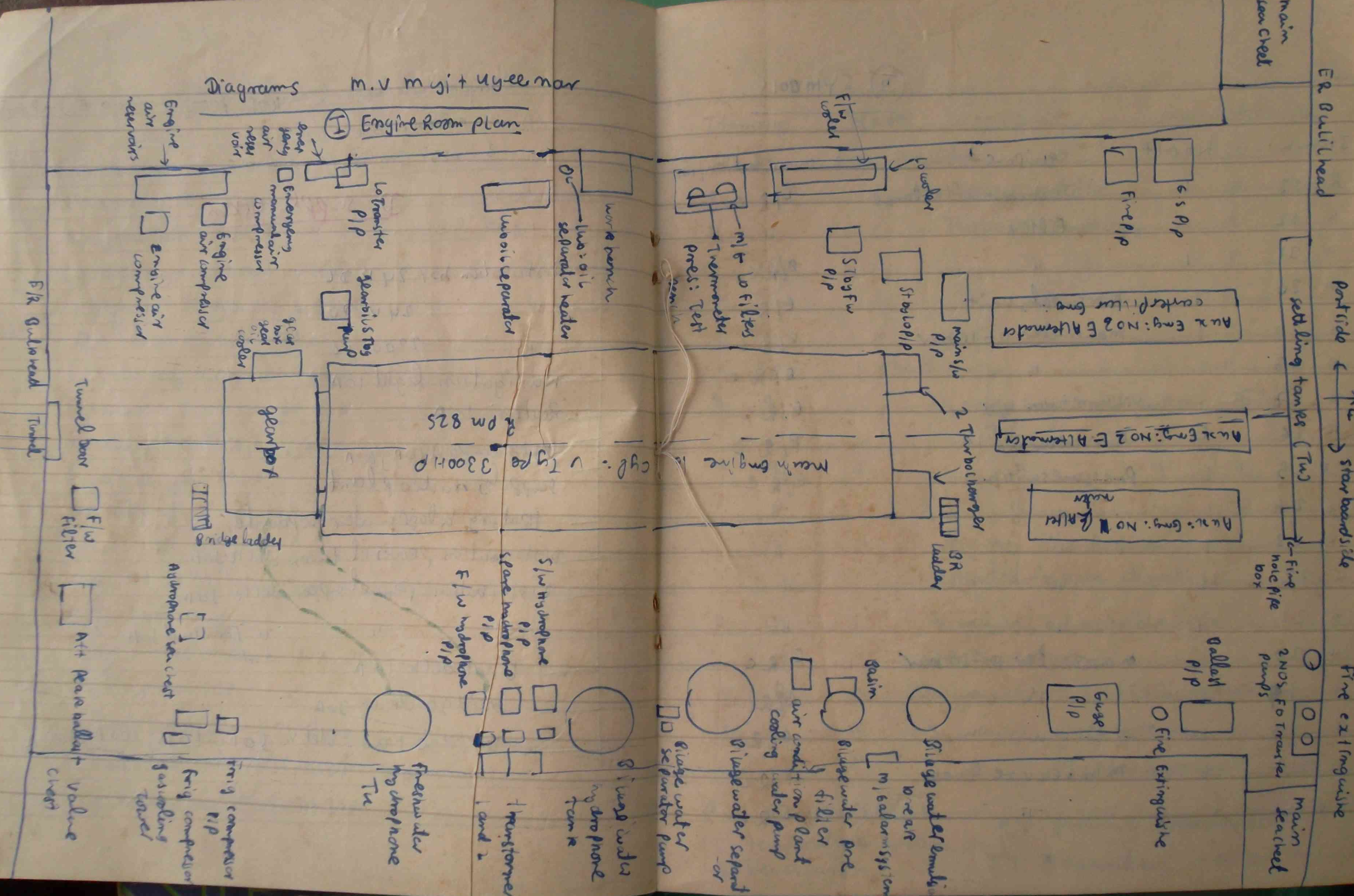
1. လျှပ်စစ်ဂဟေ ဆော (အပူချိန် 7000° – 10000° F) ရှိသည်။
2. ဓာတ်ငွေ့ ဂဟေဆော်
3. ကျွမ်း ဖြတ်ခြင်း

DC supply system

- Distribution box 24V DC
- " 24V DC
- " 220V DC
- Navigation light 10A
- Radar 10A
- Gyro compass 10A, 60A
- supply radio pland
- Heating bilage water separater
- Distribution pennel tween deck 50A
- Distribution pennel upper deck 50A
- " " " forward 50A
- boat deck 50A
- bridge deck 30A
- bridge Nav. deck 50A, 10A, 100A, 100A

Diagrams M.V Myit Uyee nar

(1) Engine Room plan



(II) SYMBOLS

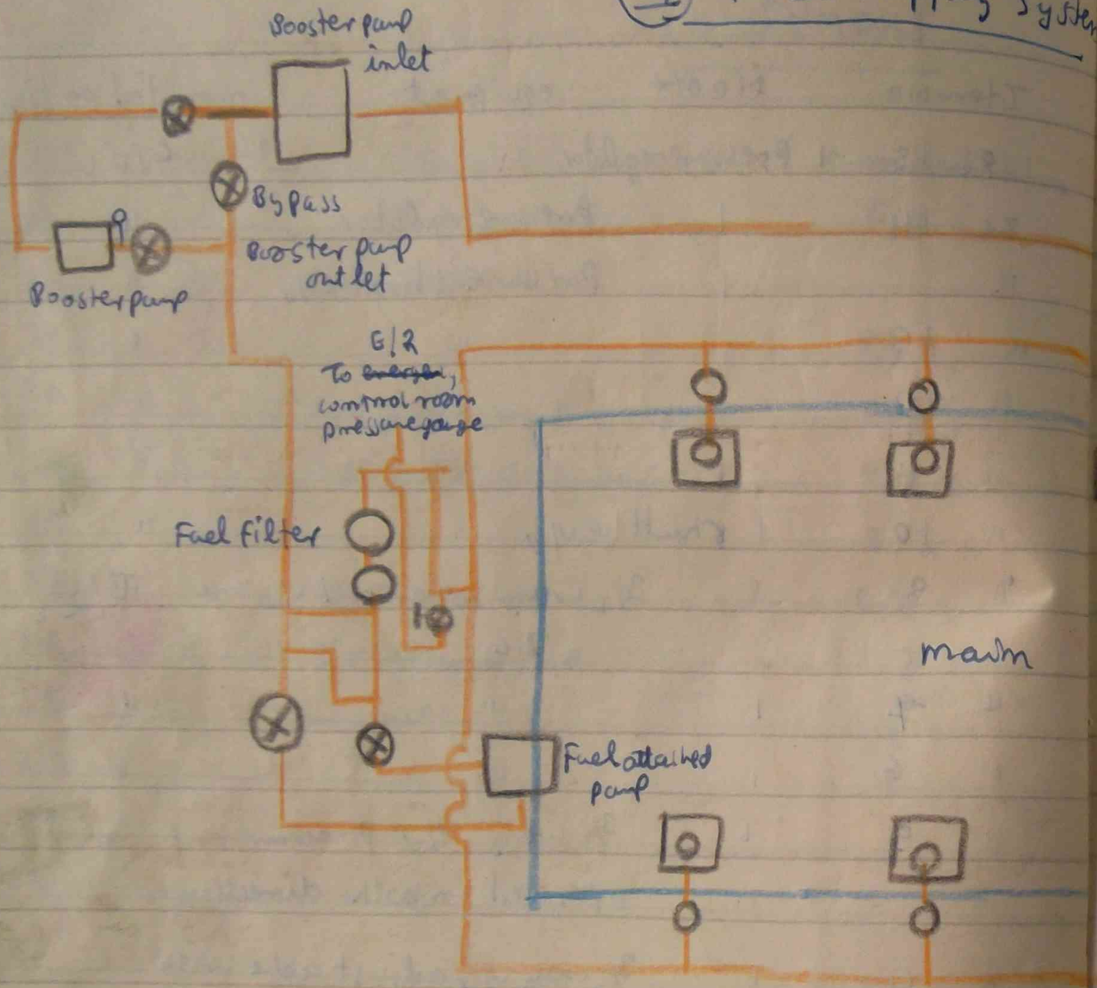
Item NU	NO off	equipment	mounted on/in
Rc 33	1	Test cock with flange	E/R C
" 32	1	Filter	"
" 31	1	"	B/R C
" 30	1	Air supply unit	E/R
" 29	1	manometer 0-10 bar	E/R C
" 28	1	"	E/R C
" 27	1	Non return v/v	G/R C
" 26	1	Air reservoir 10	B/R C
" 25	1	Pressure switch	E/R C
" 24	1	"	"
" 23	1	"	"
" 22	1	"	"
" 21	1	"	"
" 20	1	manometer 0-10 bar	E/R C
" 19	1	"	G/R1
" 18	1	"	G/R1
" 17	1	manoeuvre lever	B/R C
" 16	1	"	E/R C

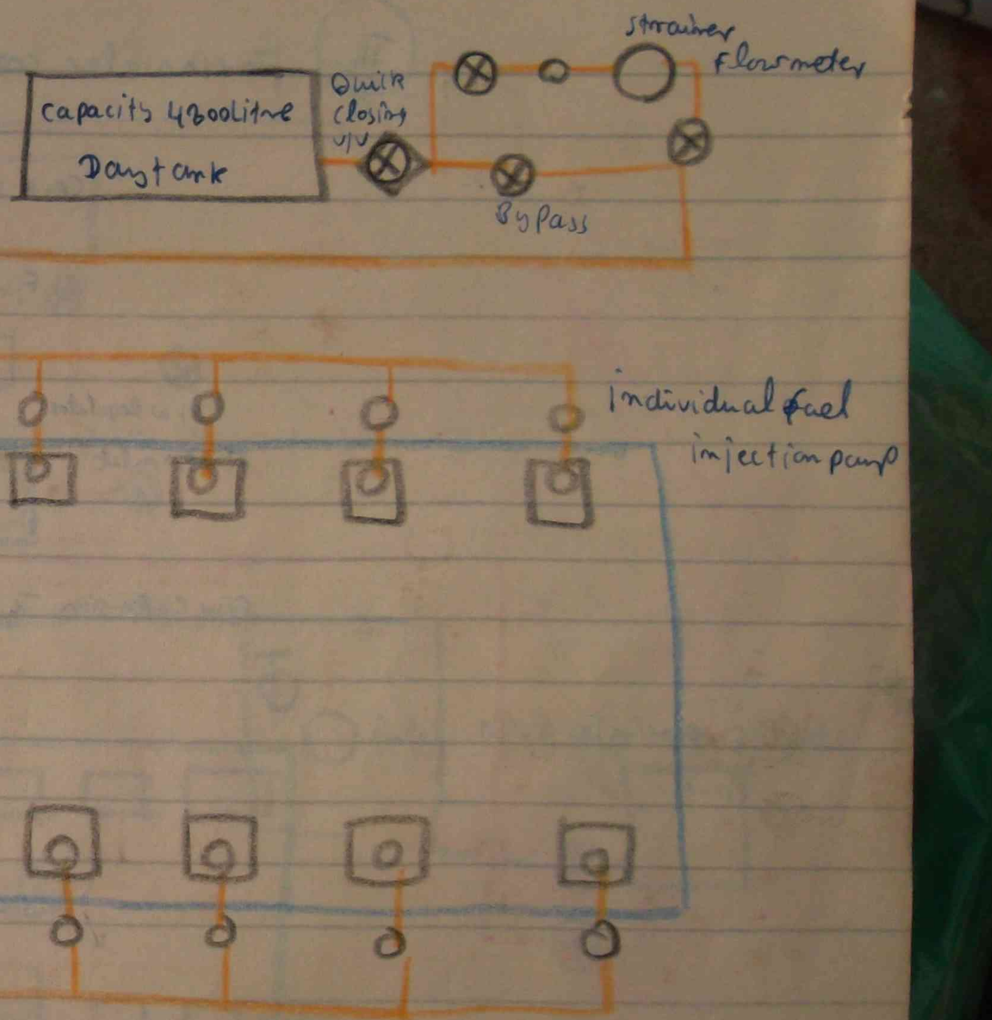
Item NU	NO off	equipment	mounted on/in
Rc 15	1	Pressure regulator	E/R C
Rc 14	1	Pressure regulator	"
"	1	Pressure throttle v/v	"
" 13	1	"	"
" 12	1	"	"
" 11	1	"	"
" 10	1	shuttle v/v	"
" 8	1	3/2 way solenoid v/v 24V 2C	TE/R C
" 7	1	" " "	"
" 6	1	" " "	"
" 5	1	3/2 way v/v pneum operated in both direction	"
" 4	1	3/2 way v/v adjustable with spring return	"
" 3	"	3/2 way v/v with spring return	"
" 2	"	3/2 way v/v red bottom	"
" 1	"	" green bottom	"

PS pressure switch
TS temperature switch
LS Level switch

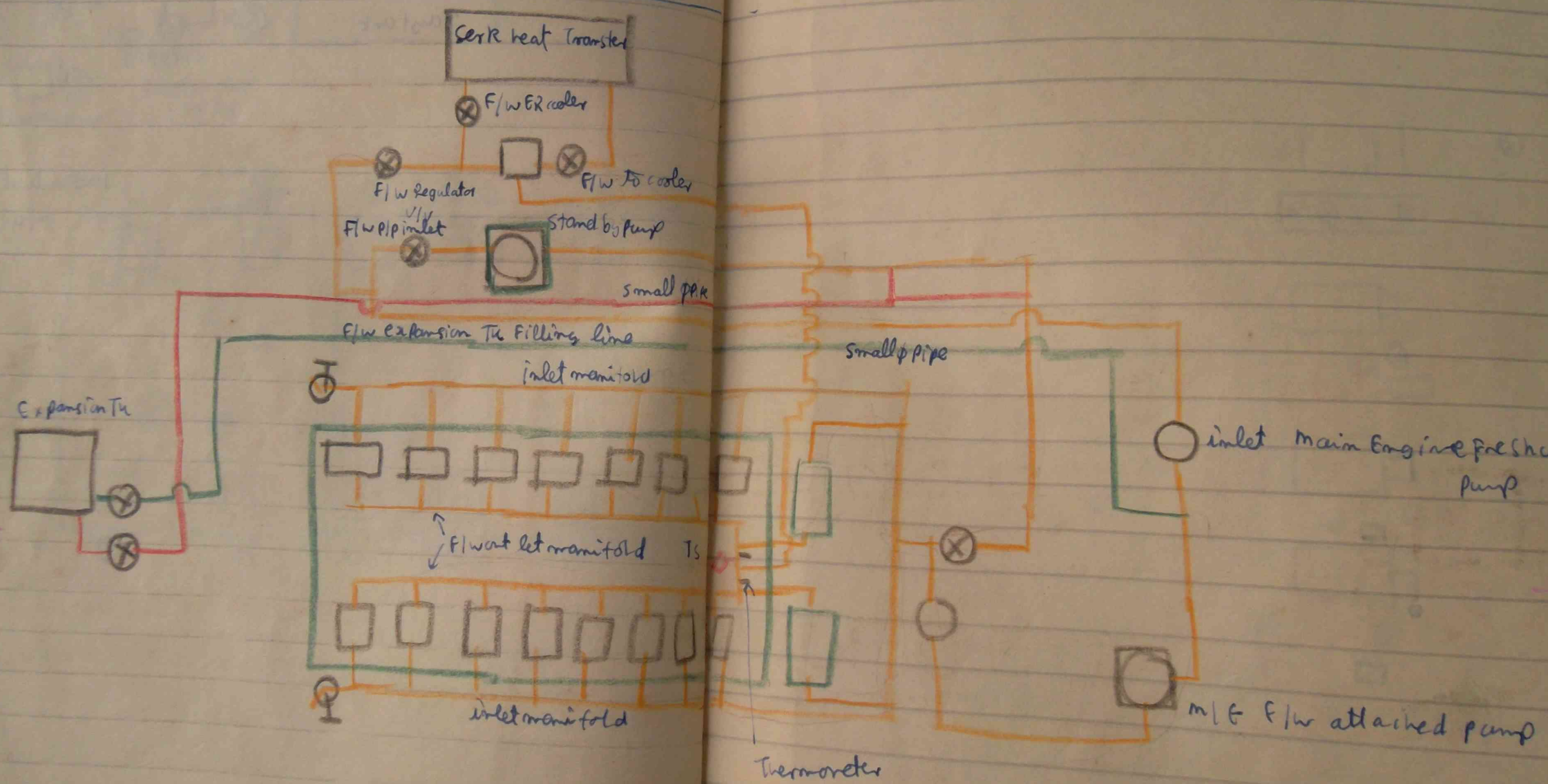
PT Pressure transmitter
TT Temperature transmitter.

III Fuel supply system

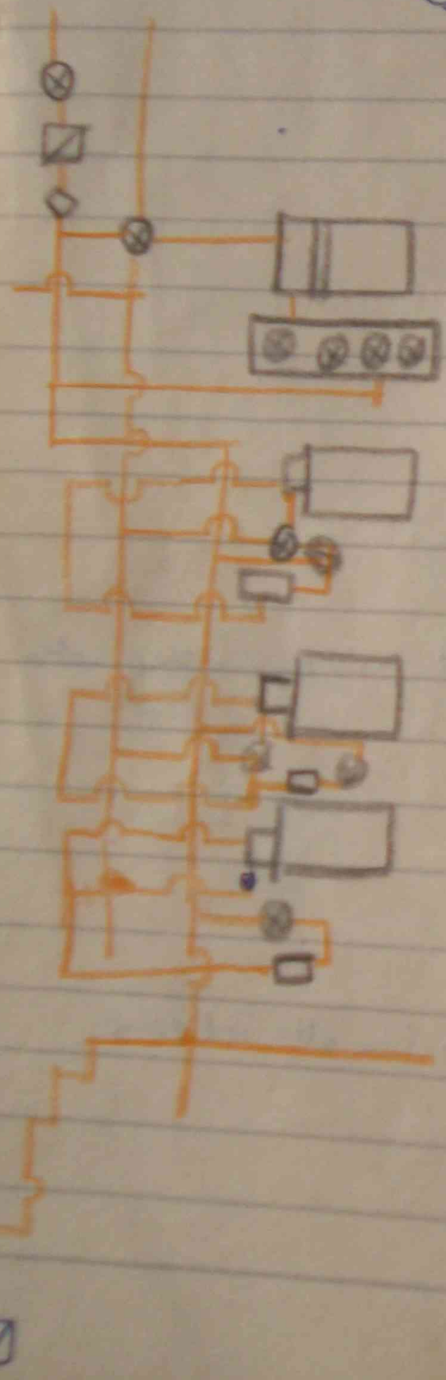




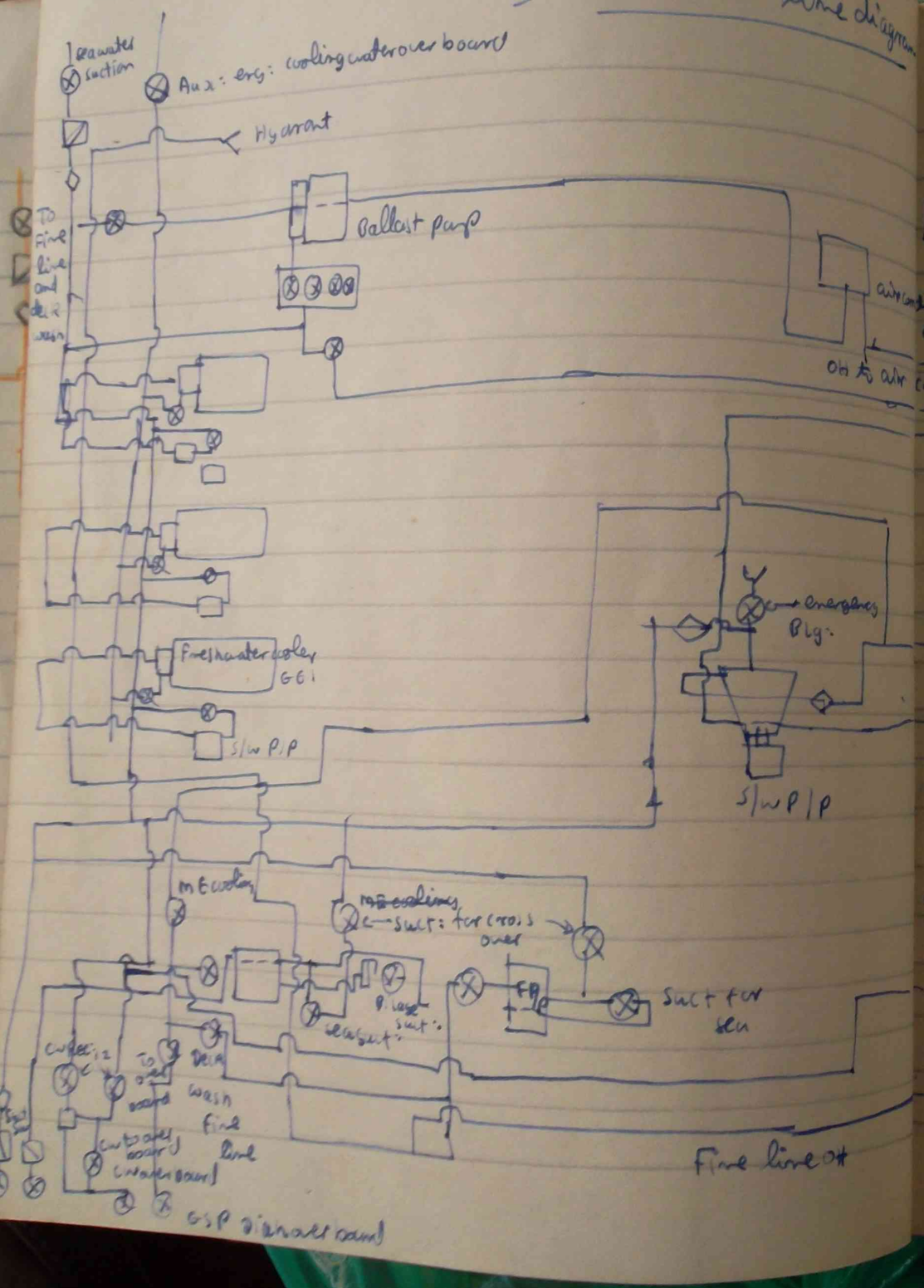
(III) Freshwater cooling line diagram



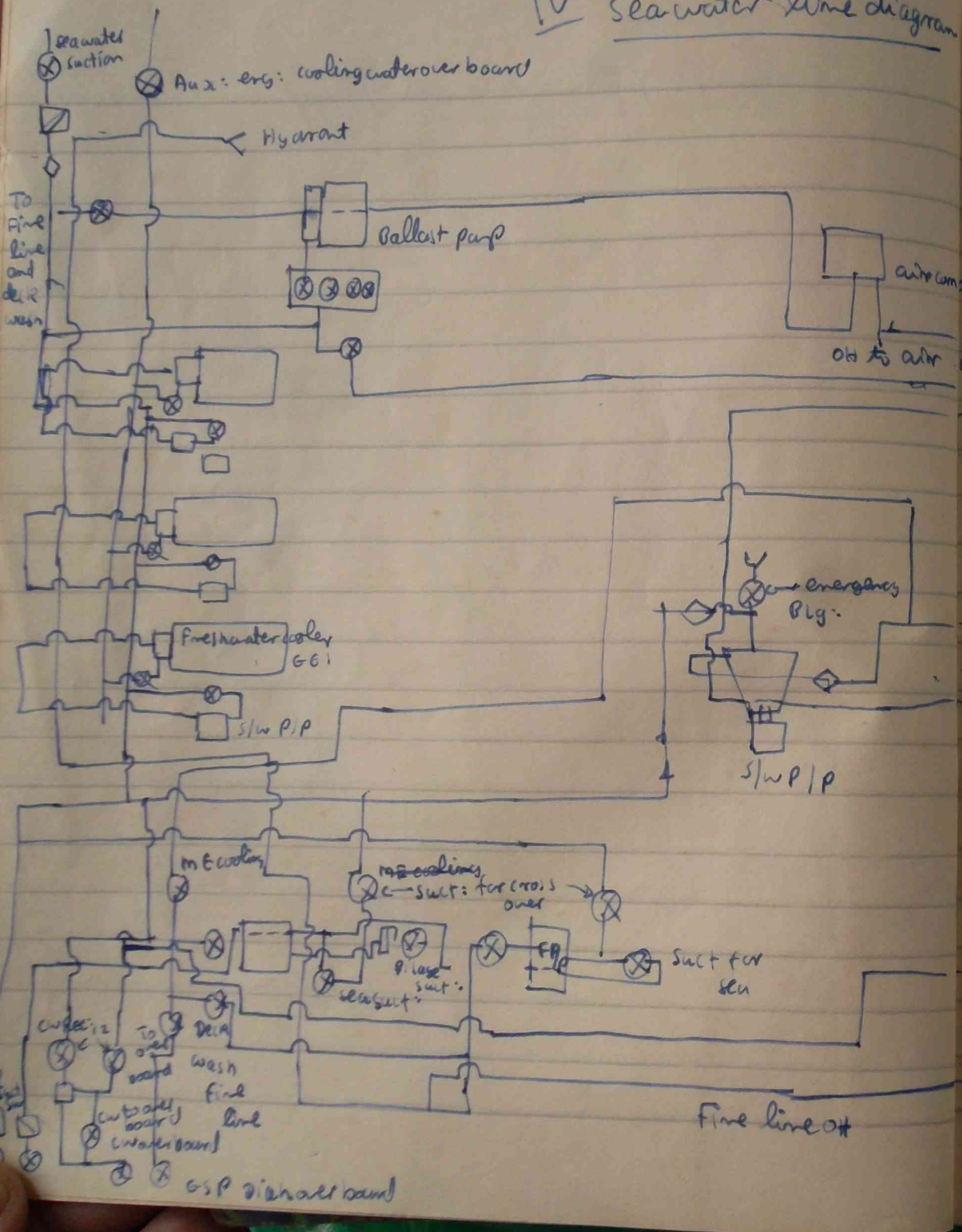
(IV) sea water line diagram

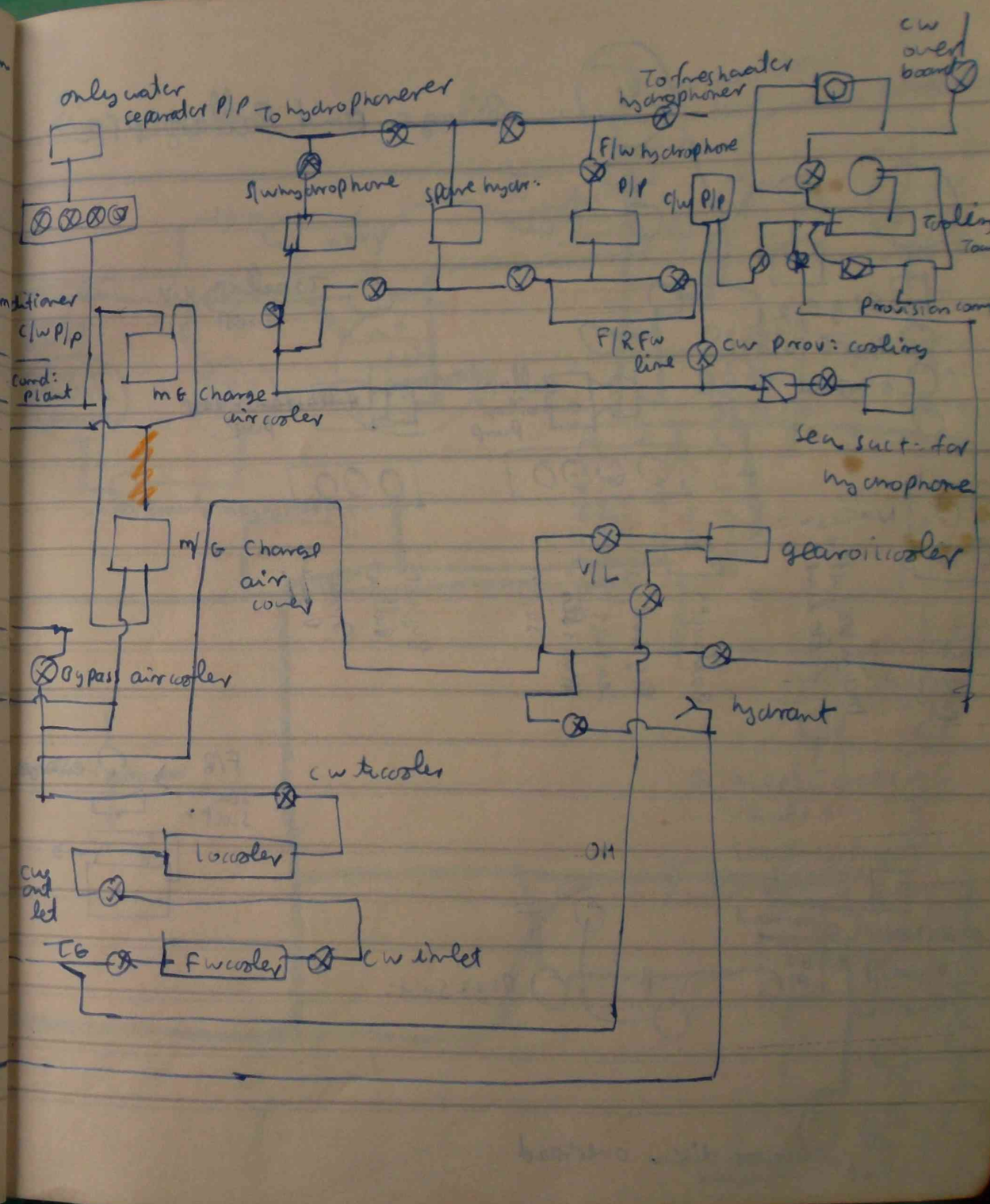


IV Seawater line diagram

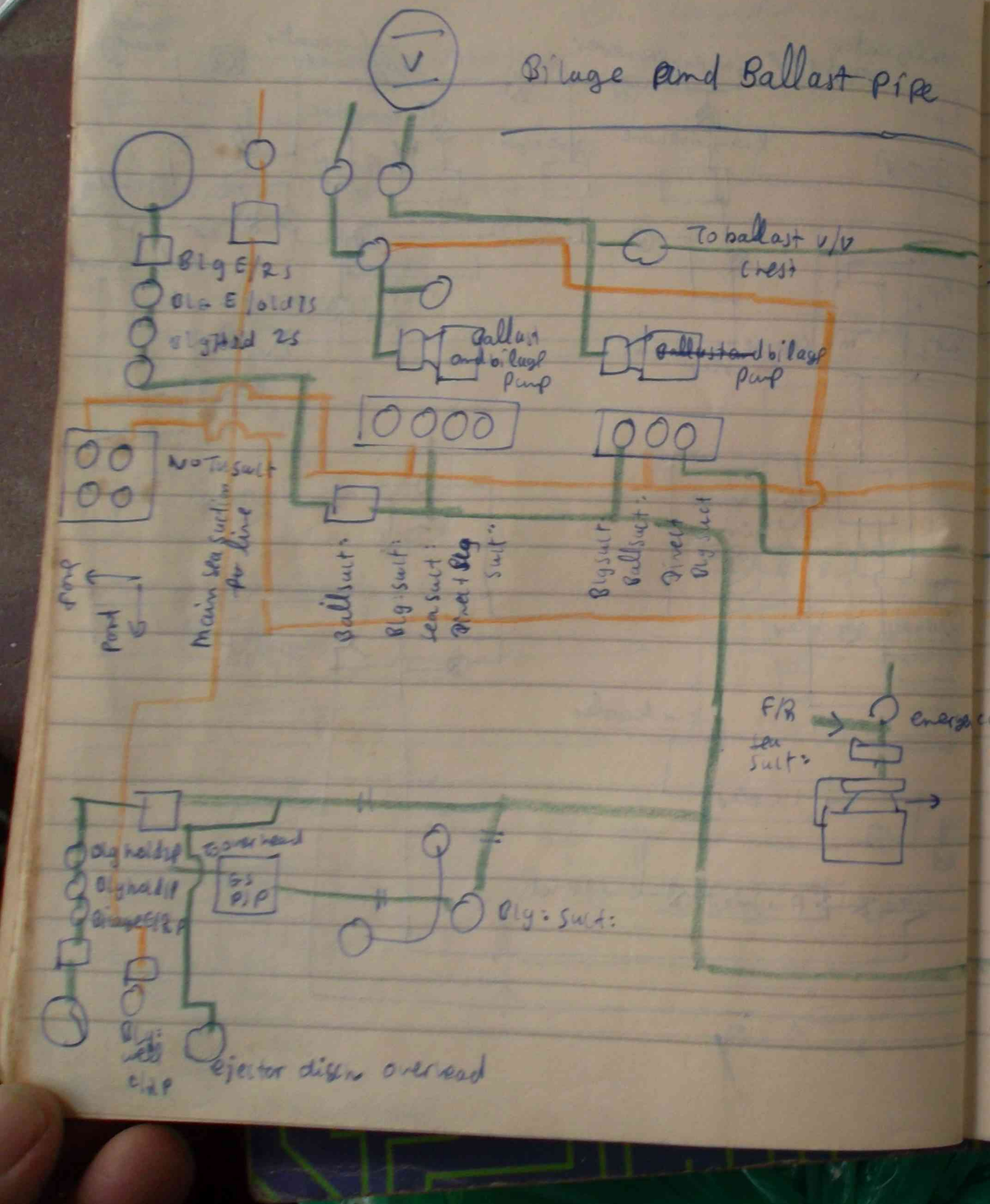


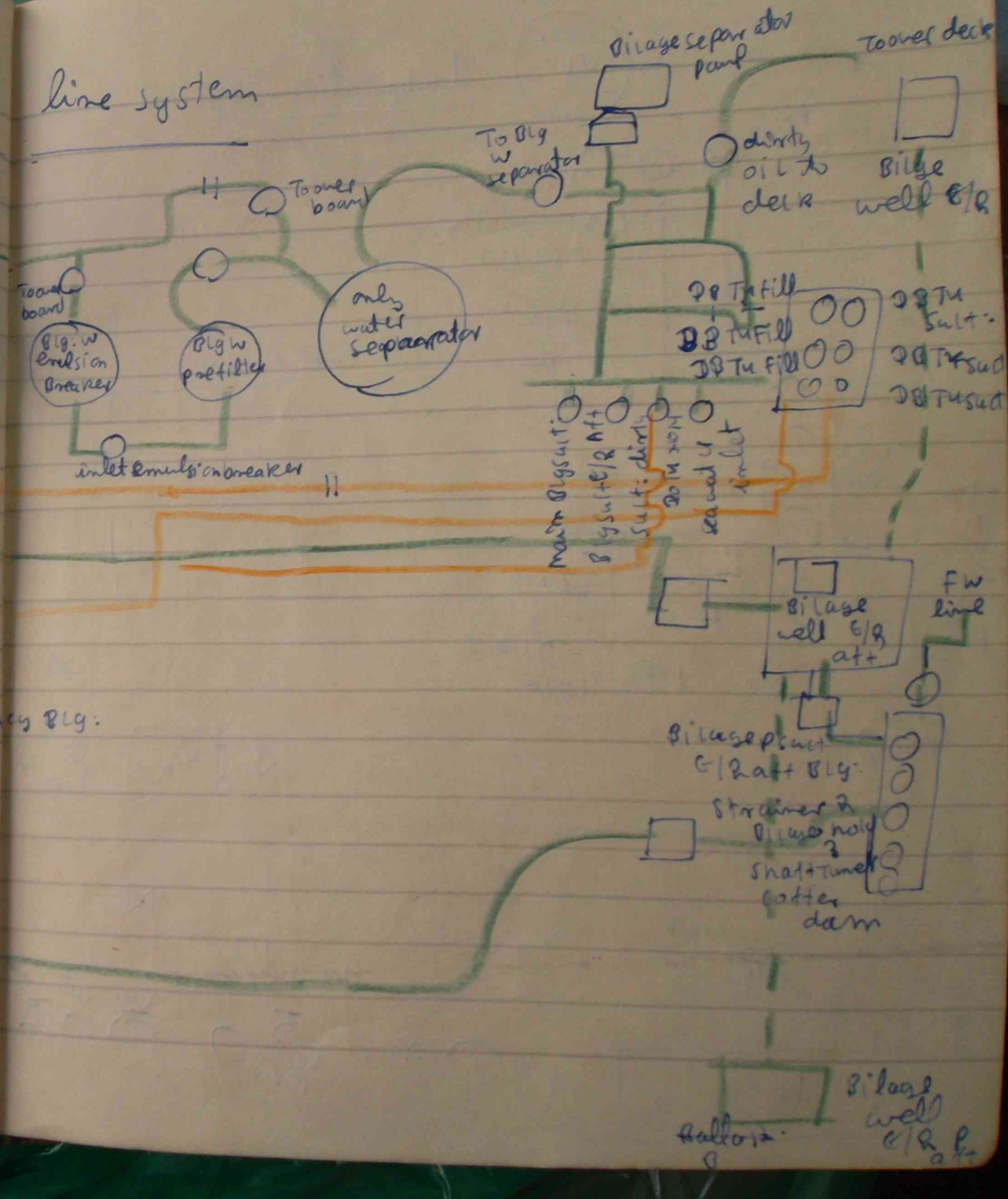
IV Sea water line diagram



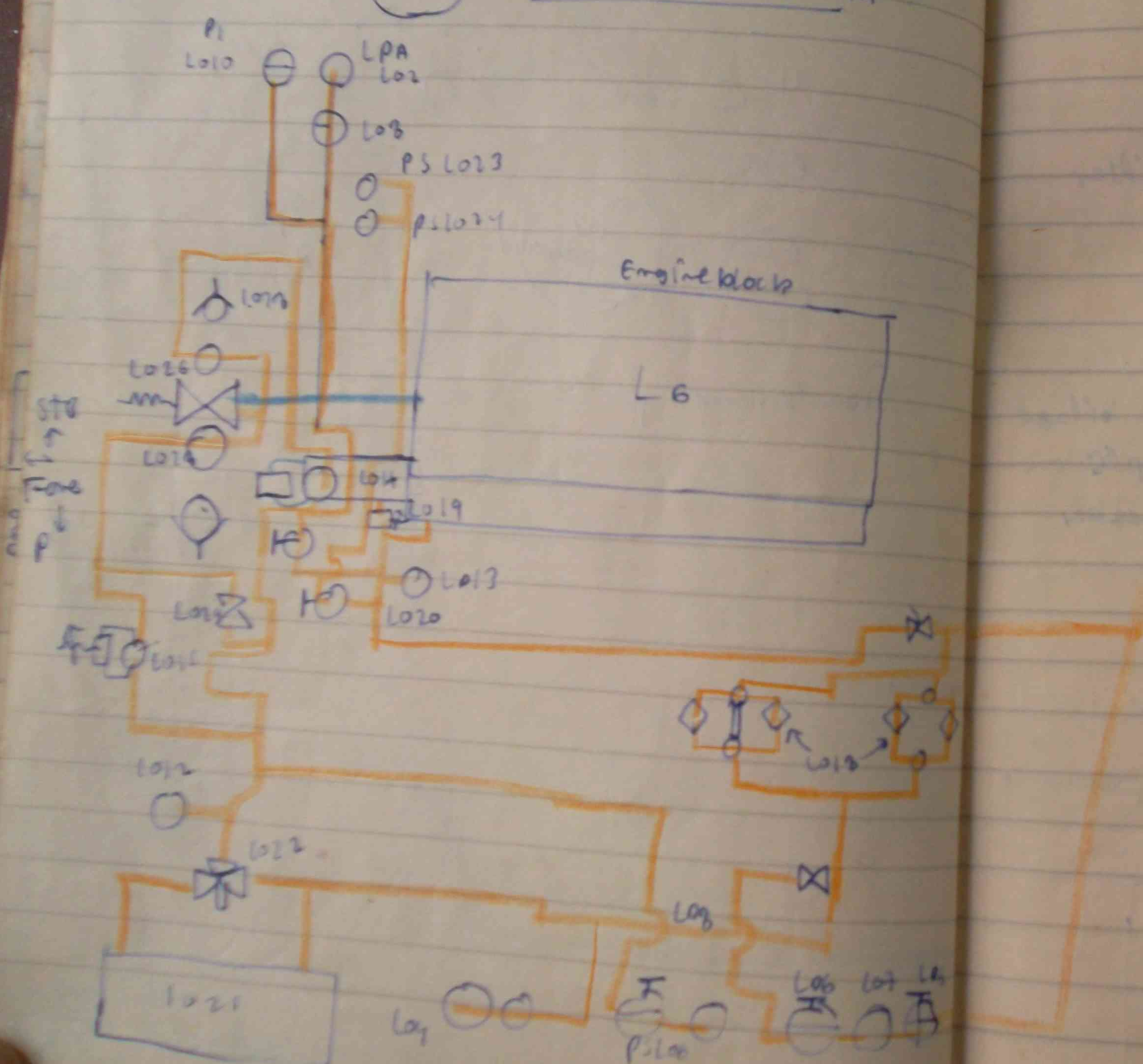


Bilge and Ballast Pipe line system





VI Lubrication system



LO 29	switch value	portside front of eng:
LO 28	3 ways v/v	"
LO 27		
LO 26	Expansion bellows	
LO 25		
LO 24	PS blocking of remote start	under of air charge cooler of Starboard
LO 23	PS Automatic of remote start	"
LO 22	Thermostatic v/v	
LO 21	oil cooler	
LO 20	Test cock with flange	front part of Eng: Portside
LO 19	Press regulating v/v	
LO 18	LO Filter	
LO 17		
LO 16		
LO 15	LO pump (standby, with motor)	
LO 14	LO pump (pressure)	Front Part of Eng: Portside
LO 13	Thermometer	
LO 12	"	
LO 11 }	manometer	
LO 10 }		
LO 9 }		

Lo 8 Test circuit w flange

Lo 7 Psalarm ~~off~~ diff Press filter oPD above lo filter

Lo 6 Ps start st by Lo Pipe

Lo 5

Lo 4 Ls alarm Lo Tu (LLA)

Lo 3

Lo 2 Psalarm Lo Press (LPA) above bilge w emplsion breaker

Lo 1

VII ENGINE & BRIDGE CONSOLE

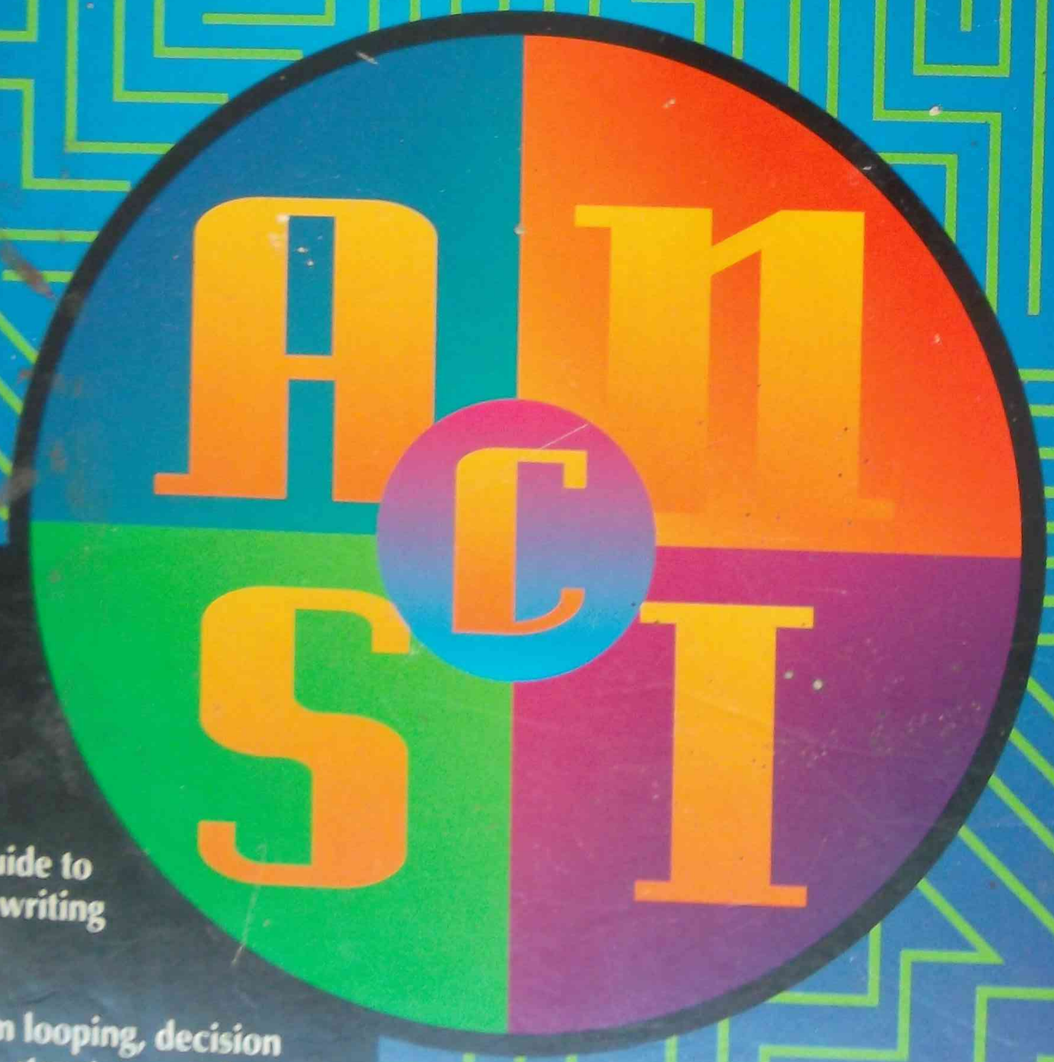
Pneumatic remote control system

Stephen G. Kochan

Programming in ANSI C

Revised Edition

The Classic Best-Seller Updated and Improved



The complete introduction to the standardize C language

Step-by-step guide to compiling and writing programs

Covers program looping, decision making, arrays, structures, pointers, and more

SAMS PUBLISHING

Contents

1

CHAPTER

Introduction

The original version of the "C" programming language was pioneered by Dennis Ritchie at AT&T Bell Laboratories in the early 1970s. It was not until the late 1970s, however, that this programming language began to gain widespread popularity and support. This was because until that time C compilers were not readily available for commercial use outside of Bell Laboratories. Initially, C's growth in popularity was also spurred on in part by the equal, if not faster, growth in popularity of the UNIX operating system. This operating system, which was also developed at Bell Laboratories, has C as its "standard" programming language. In fact, well over 90 percent of the operating system itself is written in the C language!

The enormous success of the IBM PC and its look-alikes soon made MS-DOS the most popular environment for the C language. As C grew in popularity across different operating systems, more and more vendors hopped on the bandwagon and started marketing their own C compilers. For the most part, their version of the C language was based on an appendix found in the first C programming text—*The C Programming Language*—by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). Unfortunately, this appendix did not provide a complete and unambiguous definition of C, meaning that vendors were left to interpret some aspects of the language on their own.

In the early 1980s a need was seen to standardize the definition of the C language. The American National Standards Institute (ANSI) is the organization that handles such things, so in 1983 an ANSI committee (called X3J11) was formed to standardize C.

With an ANSI definition for C, you are assured that anyone who sells a true ANSI C compiler has incorporated all of the features of the language specified by the standard. It also means that programmers who are careful can truly write C programs that will run without modification on any system that has an ANSI C compiler.

C is a "higher-level language," yet it provides capabilities that enable the user to "get in close" with the hardware and deal with the computer on a much lower level. This is because, although C is a general-purpose structured programming language, it was originally designed with systems programming applications in mind and, as such, provides the user with an enormous amount of power and flexibility. In fact, programming applications exist that could be easily handled by the C language, but that would be difficult to develop in other languages such as Pascal, FORTRAN, or BASIC.

This book proposes to teach you how to program in ANSI C. It assumes no previous exposure to the language and was designed to appeal to novice and experienced programmers alike. If you have previous programming experience, you will find that C has a unique way of doing things that probably differs significantly from any language you have used. Even if you are coming from a Pascal background—a language that C superficially resembles—you will quickly discover many features that are unique to this language, such as pointers, character strings, and bit operations.

Every feature of the C language is treated in this text. As each new feature is presented, a small *complete* program example is usually provided to illustrate the feature. This reflects the overriding philosophy that has been used in writing this book: to teach by example. Just as a picture is worth a thousand words, so is a properly chosen program example. If you have access to a computer facility that supports the ANSI C programming language, you are strongly encouraged to enter and run each program presented in this book and to compare the results obtained on your system to those shown in the text. By doing so, not only will you learn the language and its syntax, but you will also become familiar with the process of typing in, compiling, and running C programs.

The style used for teaching ANSI C is one of posing a particular problem for solution on a computer and then proceeding to develop a program in C to solve the problem. In this manner, new language constructs are introduced as they are needed to solve a particular problem.

You will find that program readability has been stressed throughout the book. This is because I strongly believe that programs should be written so that they may be easily read—either by the author or by somebody else. Through experience and common sense, you will find that such programs are almost always easier to write, debug, and modify. Furthermore, developing programs that are readable is a natural result of a true adherence to a structured programming discipline.

Because this book was written as a tutorial, the material covered in each chapter is based on previously presented material. Therefore, maximum benefit will be derived from this book by reading each chapter in succession, and you are highly discouraged from "skipping around." You should also work through the exercises that are presented at the end of each chapter before proceeding on to the next chapter.

Chapter 2, which covers some fundamental terminology about higher-level programming languages and the process of compiling programs, has been included to make sure that we are speaking the same language throughout the remainder of the text. From Chapter 3 on, you will be slowly introduced to the ANSI C language. By the time Chapter 16 rolls around, all the essential features of the language will have been covered. Chapter 16 goes into more depth about I/O operations in C. Finally, Chapter 17 includes those features of the language that are of a more advanced or esoteric nature.

Appendix A provides a complete summary of the language and is provided for reference purposes. Appendix B provides a summary of most of the standard library routines that you will find on all systems that support ANSI C. Appendix C summarizes the major differences between the ANSI C language described in this text and the "older" C. In Appendix D you'll find a list of common programming mistakes. Appendix E is a single-page ASCII chart. Finally, Appendix F contains the answers to the odd-numbered exercises found at the end of each chapter.

This book makes no assumptions about a particular computer system or operating system on which the ANSI C language is implemented. The text makes brief mention of how to compile and execute programs under MS-DOS and UNIX. The best reference for more detailed information is the documentation that came with your compiler.

I wish to thank the following people for their help in the preparation of the original text, *Programming in C*: Douglas McCormick, Maureen Connelly, Jim Scharf, Henry Tabickman, and, above all, Ken Brown. I also wish to thank

Henry Mullish of New York University for teaching me so much about writing and for getting me started in the publishing business. I'd also like to thank Dick Fritz, who has generously given comments for many books I have written, including the original and revised editions of this text.

Finally, I wish to thank my wife Leela, who provided her patience and understanding through three versions of this text.

2

CHAPTER

Some Fundamentals

This chapter describes some fundamental terms that you must understand before you can learn how to program in C. A general overview of the nature of programming in a higher-level language is provided, as is a discussion of the process of compiling a program developed in such a language.

Programming

Computers are really very dumb machines indeed, since they do only what they are told to do. Most computer systems perform their operations on a very primitive level. For example, most computers know how to add one to a number or how to test if a number is equal to zero or not. The sophistication of these basic operations usually does not go much further than that. The basic operations of a computer system form what is known as the computer's *instruction set*. Some computers have very limited instruction sets. For example, the older DEC PDP-8 computer has a very limited instruction set, while the modern DEC VAX machines contain instruction sets consisting of hundreds of basic operations.

In order to solve a problem using a computer, we must express the solution to the problem in terms of the instructions of the particular computer. A computer *program* is actually just a collection of the instructions necessary to solve a specific problem. The approach or method that is used to solve the problem is known as an *algorithm*. For example, if we wish to develop a program that tests if a number is odd or even, then the set of statements that solves the problem becomes the program. The method that is used to test if the number is even or odd is the algorithm. Normally, to develop a program to solve a particular problem, we first express the solution to the problem in terms of an algorithm and then develop a program that implements that algorithm. So the algorithm for solving the even/odd problem might be expressed as follows: "First divide the number by two. If the remainder of the division is zero, then the number is even; otherwise, the number is odd." With the algorithm in hand, we can then proceed to write the instructions necessary to implement the algorithm on a particular computer system. These instructions would be expressed in the statements of a particular computer language, such as BASIC (Beginner's Algorithmic Symbolic Interchange Code), Pascal (named after the famous mathematician Blaise Pascal), or C.

Higher-Level Languages

When computers were first developed, the only way they could be programmed was in terms of binary numbers that corresponded directly to specific machine instructions and locations in the computer's memory. The next technological software advance occurred in the development of *assembly languages*, which enabled the programmer to work with the machine on a slightly higher level. Instead of having to specify sequences of binary numbers to carry out particular tasks, the assembly language permits the programmer to use symbolic names to perform various operations and to refer to specific memory locations. A special program, known as an *assembler*, translates the assembly language program from its symbolic format into the specific machine instructions of the computer system.

Because a one-to-one correspondence still exists between each assembly language statement and a specific machine instruction, assembly languages are regarded as low-level languages. The programmer must still learn the instruction set of the particular computer system in order to write a program in assembly language, and the resulting program is not *portable*; that is, the program will not run on a different computer model without being rewritten. This is because different computer systems have different instruction sets, and since assembly language programs are written in terms of these instructions sets, they are machine dependent.

Then, along came the so-called higher-level languages, of which the FORTRAN (FORmula TRANslation) language was one of the first. Programmers developing programs in FORTRAN no longer had to concern themselves with the architecture of the particular computer, and operations performed in FORTRAN were of a much more sophisticated or higher level, far removed from the instruction set of the particular machine. One FORTRAN instruction or *statement* would result in many different machine instructions being executed, unlike the one-to-one correspondence found between assembly language statements and machine instructions.

Standardization of the syntax of a higher-level language meant that a program could be written in the language to be machine independent. That is, a program could run on any machine that supported the language with few or no changes.

In order to support a higher-level language, a special computer program must be developed that translates the statements of the program developed in the higher-level language into a form that the computer can understand—in other words, into the particular instructions of the computer. Such a program is known as a *compiler*.

Operating Systems

Before we continue with compilers, it is worthwhile to understand the role that is played by a computer program known as an *operating system*.

An operating system is a program that controls the entire operation of a computer system. All I/O operations that are performed on a computer system are channeled through the operating system. The operating system must also manage the computer system's resources and must handle the execution of programs.

One of the most popular operating systems today is the UNIX operating system, which was developed at Bell Laboratories. UNIX is a rather unique operating system in that it can be found on many different types of computer systems. Historically, operating systems were typically associated with only one type of computer system. But because UNIX is written primarily in the C language and makes very few assumptions about the architecture of the computer, it has been successfully ported to many different computer systems with a relatively small amount of effort.

Microsoft's MS-DOS is another example of a popular operating system. That system is found primarily on the IBM PC computer and compatibles.

Compiling Programs

A compiler is a software program that is, in principle, no different from the ones you will see in this book, although it is certainly much more complex. A compiler analyzes a program developed in a particular computer language and then translates it into a form that is suitable for execution on your particular computer system.

Figure 2.1 shows the steps that are involved in entering, compiling, and executing a computer program developed in the C programming language and the typical UNIX commands that are used.

The program that is to be compiled is first typed into a *file* on the computer system. Computer installations have various conventions that are used for naming files, but in general, the choice of the name is up to you. Under UNIX, C programs can be given any name provided the last two characters are ".c". So the name `prog1.c` would be a valid file name for a C program running under UNIX.

A text editor must usually be used to enter the C program into a file. Some C compiler vendors supply a text editor when you purchase their software. `vi` is a popular text editor used on UNIX systems. In order to be able to try the programs presented in this book, you will first have to learn how to use such an editor. Check the documentation that comes with your system for more information about the text editors that are available.

The program that is entered into the file is known as the *source program*, since it represents the original form of the program expressed in the C language. Once the source program has been entered into a file, you can then proceed to have it compiled.

The compilation process is initiated by typing a special command on the system. When this command is entered, the name of the file that contains the source program must also be specified. For example, under UNIX, the command to initiate program compilation is called `cc`. The `cl` command activates many of the C compilers found on PCs running MS-DOS. Typing the line

```
cl prog1.c
```

would have the effect of initiating the compilation process with the source program contained in `prog1.c`.

In the first step of the compilation process, the compiler examines each program statement contained in the source program and checks it to ensure that it conforms to the syntax and semantics of the language. If any mistakes are discovered by the compiler during this phase, then they will be reported to the user and the compilation process will end right there. The errors will then have to be corrected in the source program (with the use of the text editor), and the

compilation process restarted. Typical errors reported during this phase of compilation might be due to an expression that has unbalanced parentheses (syntactic error), or due to the use of a variable that is not "defined" (semantic error).

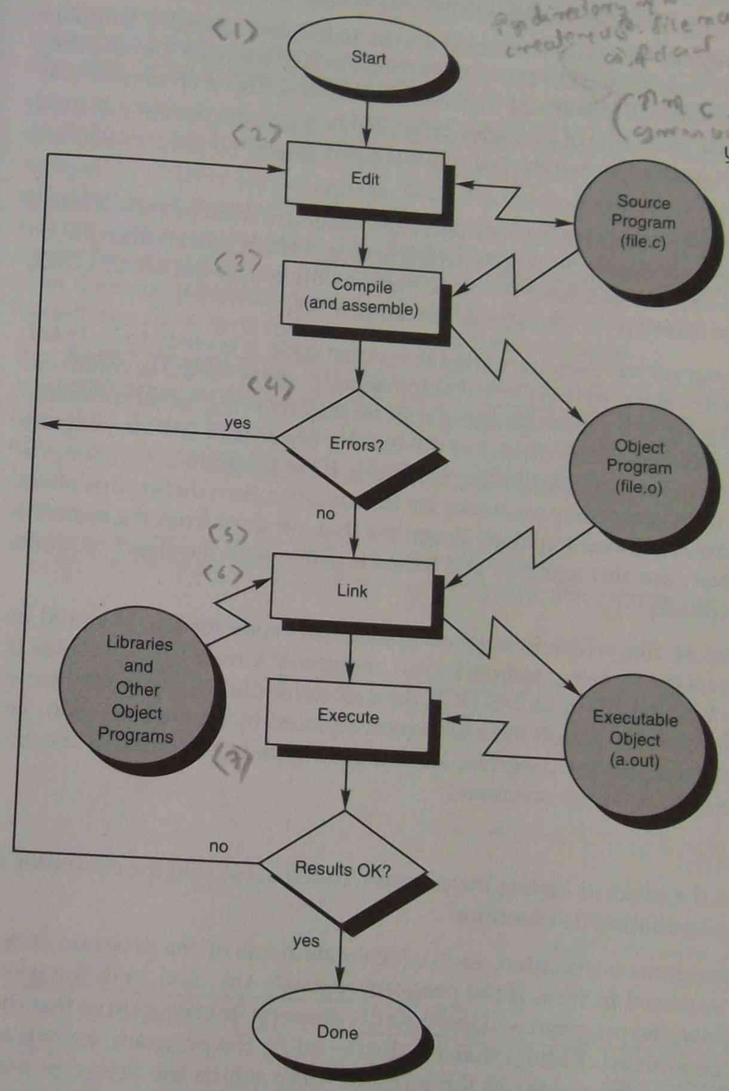


Figure 2.1. Typical steps for entering, compiling, and executing C programs.

When all of the syntactic and semantic errors have been removed from the program, the compiler will then proceed to take each statement of the program and translate it into a "lower" form. On most machines, this means that each statement will be translated by the compiler into the equivalent statement or statements in assembly language needed to perform the identical task.

After the program has been translated into an equivalent assembly language program, the next step in the compilation process is to translate the assembly language statements into actual machine instructions. This step may or may not involve the execution of a separate program known as an *assembler*. On most systems, the assembler is executed automatically as part of the compilation process.

The assembler takes each assembly language statement and converts it into a binary format known as *object code*, which is then written into another file on the system. This file will have the same name as the source file under UNIX, with the last letter an "o" (for *object*) instead of a "c".

After the program has been translated into object code, it is ready to be *linked*. This process is once again performed automatically whenever the `cc` command is issued under UNIX. Other operating systems may require another command to perform this task. The purpose of the linking phase is to get the program into a final form for execution on the computer. If the program uses other programs that were previously processed by the compiler, then during this phase the programs are linked together. Programs that are used from the system's program *library* are also searched and linked together with the object program during this phase.

The final linked file, which is in an *executable object* code format, is stored in another file on the system, ready to be run or *executed*. Under UNIX, this file is called `a.out` by default. Under MS-DOS, the executable file usually has the same name as the source file, with the `c` extension replaced by an `exe` extension. To subsequently execute the program, all you do is type in the name of the executable object file. So the command

```
a.out
```

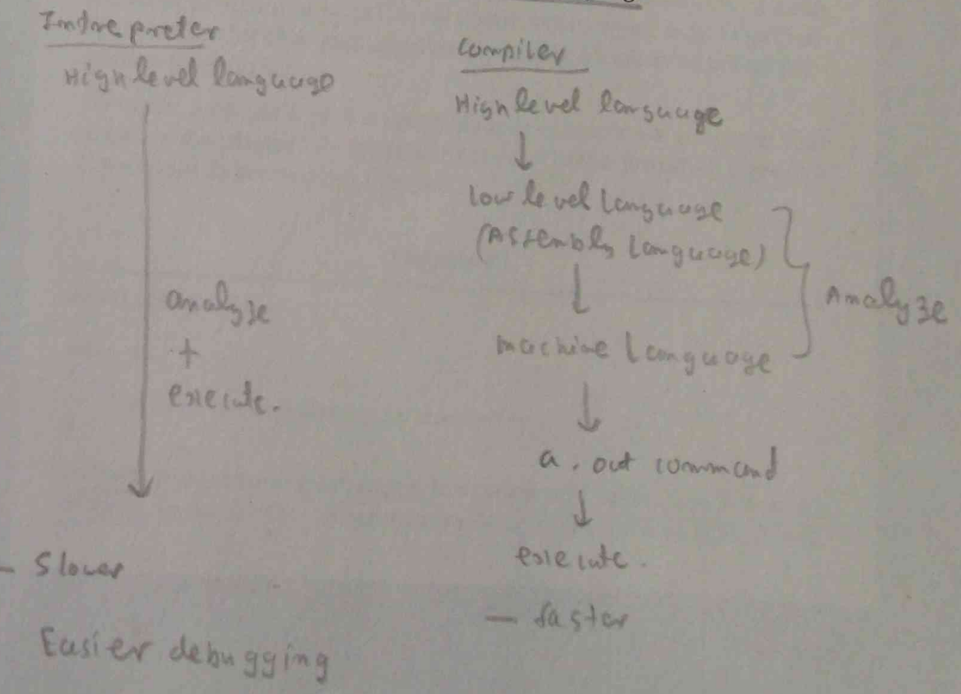
would have the effect of *loading* the program called `a.out` into the computer's memory and initiating its execution.

When the program is executed, each of the statements of the program is sequentially executed in turn. If the program requests any data from the user, known as *input*, the program will temporarily suspend its execution so that the input may be entered. Results that are displayed by the program, known as *output*, will normally appear at the terminal from which the program was executed.

If all goes well (and it probably won't the first time the program is executed), the program will perform its intended functions. If the program does not produce the desired results, then it will be necessary to go back and reanalyze the program's logic. This is known as the *debugging phase*, during which an attempt is made to remove all the known problems or *bugs* from the program. In order to do so, it will most likely be necessary to make changes to the original source program. In that case, the entire process of compiling, linking, and executing the program must be repeated until the desired results are obtained.

Before leaving this discussion of the compilation process, we should point out that there is another method used for analyzing and executing programs developed in a higher-level language. With this method, programs are not compiled but are *interpreted*. An interpreter analyzes and executes the statements of a program at the same time. This method usually allows programs to be more easily debugged. On the other hand, interpreted languages are typically slower than their compiled counterparts, since the program statements are not converted into a low-level form in advance of their execution.

The BASIC programming language is the most popular language in which programs are typically interpreted and not compiled. Other examples include LISP (LISt Processing), the UNIX system's *shell*, and PostScript. Some vendors also offer interpreters for the C programming language.



CHAPTER 3

Writing a Program in ANSI C

In this chapter you are introduced to the C language so that you can see what programming in C is all about. What better way to gain an appreciation for this language than by taking a look at an actual program written in C?

To begin with, let's pick a rather simple example—a program that displays the phrase "Programming is fun" at the terminal. Here is a C program to accomplish this task.

Program 3.1

```
#include <stdio.h>

main ()
{
    printf ("Programming is fun.\n");
}
```

In the C programming language, lowercase and uppercase letters are distinct. In most other programming languages, such as FORTRAN,

COBOL (COmmon Business Oriented Language), PL/I (Programming Language I), or BASIC, uppercase letters are used exclusively. If you are used to programming in any of these languages, the use of lowercase letters may take some getting used to.

Also, unlike many other programming languages, C does not particularly care where on the line you begin typing. Some languages, such as FORTRAN and COBOL, are very fussy about such things, but in C you may begin typing your statement at any position on the line. This fact can be used to your advantage in developing programs that are easier to read.

Returning to our first C program, if we were to type this program into the computer, issue the proper commands to the system to have it compiled, and then execute it, we could expect the following results or output to appear at the terminal:

```
Programming is fun.
```

Let's take a closer look at our first program. The first line of the program

```
#include <stdio.h>
```

should be included at the beginning of just about every program you write. It tells the compiler information about the `printf` output routine that is used later in the program. We'll talk in detail about what this line does in Chapter 13, "The Preprocessor."

QUICK TIP

Remember to put this special `include` statement at the beginning of all your programs and to type it in precisely as shown in the text.

The line of the program that reads

```
main ()
```

informs the system that the name of the program is `main`. `main` is a special name that indicates precisely *where* the program is to begin execution. The open and closed parentheses immediately following `main` specify that `main` is the name of a *function*. (This concept will be explained in more detail in Chapter 8, "Functions.")

Now that we have identified `main` to the system, we are ready to specify precisely what this routine is to perform. This is done by enclosing all program statements of the routine within a pair of curly braces. (For those readers who

are familiar with the Pascal programming language, the braces in C are somewhat analogous to the `Begin` and `End` block declarators of that language.) All program statements included between the braces will be taken as part of the `main` routine by the system. In Program 3.1, we have only one such statement. This statement specifies that a routine named `printf` is to be invoked or *called*. The parameter or *argument* to be passed to the `printf` routine is the string of characters

```
"Programming is fun.\n"
```

The `printf` routine is a function in the C library that simply prints or displays its argument (or arguments, as you will see shortly) at the terminal or on the video screen. The last two characters in the string, namely the backslash (\) and the letter `n`, are known collectively as the *newline* character. A newline character tells the system to do precisely what its name implies—that is, go to a new line. Any characters to be printed after the newline character will then appear on the next line of the display. In fact, the newline character is similar in concept to the carriage return key on a typewriter.

All program statements in C *must* be terminated by a semicolon (;). This is the reason for the semicolon that appears immediately following the closed parenthesis of the `printf` call.

QUICK TIP

Remember to place a semicolon at the end of all your C statements.

Now that we have finished discussing our first program, why don't we modify to also display the phrase "And programming in C is even more fun." This can be done by the simple addition of another call to the `printf` routine, as shown in Program 3.2. Remember that every C program statement must be terminated by a semicolon.

Program 3.2

```
#include <stdio.h>

main ()
{
    printf ("Programming is fun.\n");
    printf ("And programming in C is even more fun.\n");
    Return 0;
}
```

If we type in Program 3.2 and then compile and execute it, we can expect the following output at our terminal:

Program 3.2 OUTPUT

```
Programming is fun.
And programming in C is even more fun.
```

As you will see from the next program example, it is not necessary to make a separate call to the `printf` routine for each line of output. Study the program listed below in Program 3.3 and try to predict the results before examining the output (no cheating now!).

Program 3.3

```
#include <stdio.h>

main ()
{
    printf ("Testing...\n..1\n...2\n....3\n");
}
```

Program 3.3 OUTPUT

```
Testing...
..1
...2
....3
```

The `printf` routine is the most commonly used routine in this book. This is because it provides an easy and convenient means to display program results. Not only can simple phrases be displayed, but the values of *variables* and the results of computations can be as well. In fact, Program 3.4 uses the `printf` routine to display the results of adding two numbers, namely 50 and 25.

Program 3.4

```
#include <stdio.h>

main ()
{
    int sum;

    sum = 50 + 25;
    printf ("The sum of 50 and 25 is %i\n", sum);
}
```

Program 3.4 OUTPUT

```
The sum of 50 and 25 is 75
```

A bit of explanation is in order with respect to the above program. The first C program statement *declares* the *variable* `sum` to be of type *int*eger. C, like Pascal—and unlike FORTRAN or BASIC—requires that all program variables be declared before they are used in a program. The declaration of a variable specifies to the C compiler how a particular variable will be used by the program. This information is needed by the compiler in order to generate the correct instructions to store and retrieve values into and out of the variable. A variable declared as type `int` can only be used to hold integral values; that is, values without decimal places. Examples of integral values are 3, 5, -20, and 0. Numbers with decimal places, such as 3.14, 2.455, and 27.0, for example, are known as *floating point* or *real* numbers.

QUICK TIP

Remember that in C, all variables must be declared before they are used.

The integer variable `sum` will be used to store the result of the addition of the two integers 50 and 25. We have intentionally left a blank line following the declaration of this variable. This is done to visually separate the variable declarations of the routine from the program statements and is strictly a matter of style. Sometimes the addition of a single blank line in a program can help to make the program more readable.

The program statement

```
sum = 50 + 25;
```

reads as it would in most other programming languages: the number 50 is added (as indicated by the plus sign) to the number 25 and the result is stored (as indicated by the *assignment operator*, the equals sign) into the variable sum.

The printf routine call in Program 3.4 now has two items or *arguments* enclosed within the parentheses. These arguments are separated by a comma. The first argument to the printf routine is always the character string to be displayed. However, along with the display of the character string, we may frequently wish to have the value of certain program variables displayed as well. In our case, we would like to have the value of the variable sum displayed at the terminal after the characters

```
The sum of 50 and 25 is
```

are displayed. The percent character inside the first argument is a special character recognized by the printf function. The character that immediately follows the percent sign specifies what *type* of value is to be displayed at that point. In the above program, the letter i is recognized by the printf routine as signifying that an integer value is to be displayed.[1]

Whenever the printf routine finds the %i characters inside a character string, it will automatically display the value of the next argument to the printf routine. Since sum is the next argument to printf, its value is automatically displayed after the characters "The sum of 50 and 25 is" are displayed.

Now try to predict the output from Program 3.5.

Program 3.5

```
#include <stdio.h>

main ()
{
    int  value1, value2, sum;

    value1 = 50;
    value2 = 25;
    sum = value1 + value2;
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);
}
```

[1]*Note that* printf *also allows you to specify* %d *format characters to display an integer. We'll stick to* %i *throughout this book.*

Program 3.5 OUTPUT

```
The sum of 50 and 25 is 75
```

The first program statement declares three variables called value1, value2, and sum all to be of type int. This statement could have equivalently been expressed using three separate declaratory statements as follows:

```
int value1;
int value2;
int sum;
```

After the three variables have been declared, the program assigns the value 50 to the variable value1 and then 25 to value2. The sum of these two variables is then computed and the result assigned to the variable sum.

The call to the printf routine now contains four arguments. Once again, the first argument, commonly called the *format string*, describes to the system how the remaining arguments are to be displayed. The value of value1 is to be displayed immediately following the display of the characters "the sum of." Similarly, the values of value2 and sum are to be printed at the appropriate points as indicated by the next two occurrences of the %i characters in the format string.

The last program in this chapter (Program 3.6) introduces the concept of the *comment*. A comment statement is used in a program to document a program and to enhance its readability. As you will see from the following example, comments serve to tell the reader of the program—the programmer or someone else whose responsibility it is to maintain the program—just what the programmer had in mind when he or she wrote a particular program or a particular sequence of statements.

Program 3.6

```
/* This program adds two integer values and displays
   the results                                          */
#include <stdio.h>
main ()
{
    /* Declare variables */
    int  value1, value2, sum;

/* Assign values and compute the result */
    value1 = 50;
    value2 = 25;
    sum = value1 + value2;
```

continues

Program 3.6 Continued

```
    /* Display the result */
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);
}
```

Program 3.6 OUTPUT

```
The sum of 50 and 25 is 75
```

A comment statement is initiated by the two characters / and *. These two characters must be written without any intervening spaces. To end a comment statement, the characters * and / are used, once again without any embedded spaces. All characters included between the opening /* and the closing */ are treated as part of the comment statement and are ignored by the C system. In Program 3.6, four separate comment statements were used. This program is otherwise identical to Program 3.5. Admittedly, this is a contrived example, since only the first comment at the head of the program is useful. (Yes, it is possible to insert so many comments into a program that the readability of the program is actually degraded instead of improved!)

The intelligent use of comment statements inside a program cannot be overemphasized. Many times a programmer will return to a program that he coded perhaps only six months ago, only to discover to his dismay that he cannot for the life of him remember the purpose of a particular routine or of a particular group of statements. A simple comment statement judiciously inserted at that particular point in the program might have saved a significant amount of time otherwise wasted on rethinking the logic of the routine or set of statements.

It is a good idea to get into the habit of inserting comment statements into the program as the program is being written or typed into the computer. There are good reasons for this. First, it is far easier to document the program while the particular program logic is still fresh in your mind than it is to go back and rethink the logic after the program has been completed. Second, by inserting comments into the program at such an early stage of the game, you to reap the benefits of the comments during the debug phase, when program logic errors are being isolated and debugged. A comment can not only help you read through the program, but it can also help point the way to the source of the logic mistake. Finally, I have yet to discover a programmer who actually enjoyed documenting a program. In fact, once you have finished debugging your program, you will probably not relish the idea of going back to the program to

insert comments. Inserting comments while developing the program will make this sometimes tedious task a bit easier to swallow.

This concludes this introductory chapter on developing programs in C. By now you should have a good feel as to what is involved in writing a program in C and you should be able to develop a small program on your own. In the next chapter, we will begin to examine some of the finer intricacies of this wonderfully powerful and flexible programming language. But first, try your hand at the exercises that follow to make sure you understand the concepts presented in this chapter.

Exercises

1. If you have access to a computer facility that supports the C programming language, type in and run the six programs presented in this chapter. Compare the output produced by each program with the output presented after each program.

2. Write a program that prints the following text at the terminal.

   1. In C, lowercase letters are significant.
   2. `main` is where program execution begins.
   3. Open and closed braces enclose program statements in a routine.
   4. All program statements must be terminated by a semicolon.

3. What output would you expect from the following program?

```
#include <stdio.h>

main ()
{
   printf ("Testing...");
   printf ("....1");
   printf ("...2");
   printf ("..3");
   printf ("\n");
}
```

4. Write a program that subtracts the value 15 from 87 and displays the result, together with an appropriate message, at the terminal.

5. Identify five syntactic errors in the following program. Then type in and run the corrected program to make sure you have correctly identified all the mistakes.

```
#include <stdio.h>
```

```
main ()
{
        INT  sum;
        /* COMPUTE RESULT
        sum = 25 + 37 - 19
        /* DISPLAY RESULTS */
        printf ("The answer is %i\n" sum);
}
```

6. What output might you expect from the following program?

```
#include <stdio.h>

main ()
{
   int answer, result;
   answer = 100;
   result = answer - 10;
   printf ("The result is %i\n", result + 5);
}
```


4

CHAPTER

Variables, Data Types, and Arithmetic Expressions

In this chapter, we will discuss the formation of variable names and constants, take a look at the basic data types, and describe some fundamental rules for forming arithmetic expressions in C.

Variables

Early computer programmers had the onerous task of having to write their programs in the binary language of the machine they were programming. This meant that computer instructions had to be hand-coded into binary numbers by the programmer before they could be entered into the machine. Furthermore, the programmer had to explicitly assign and reference any storage locations inside the computer's memory by a specific number or memory address.

Modern-day programming languages allow the programmer to concentrate more on solving the particular problem at hand than worrying about specific machine codes or memory locations. They enable us to assign symbolic names, known as *variable names*, for storing program computations and results. A variable name can be chosen by the programmer in a meaningful way to reflect the type of value that is to be stored in that variable.

In Chapter 3, "Writing a Program in ANSI C," we used several variables to store integer values. For example, we used the variable `sum` in Program 3.4 to store the result of the addition of the two integers 50 and 25.

The C language allows data types other than just integers to be stored in variables as well, provided the proper declaration for the variable is made *before* it is used in the program. Variables can be used to store floating-point numbers, characters, and even *pointers* to locations inside the computer's memory.

The rules for forming variable names are quite simple: They must begin with a letter or underscore (_) and may be followed by any combination of letters (upper- or lowercase), underscores, or the digits 0-9. The following is a list of valid variable names.

```
sum
piece_flag
i
J5x7
Number_of_moves
_sysflag
```

On the other hand, the following variable names are not valid for the stated reasons:

`sum$value`	$ not a valid character
`piece flag`	Embedded spaces not permitted
`3Spencer`	Can't start with a number
`int`	Reserved word

`int` cannot be used as a variable name since its use has a special meaning to the C compiler. This use is known as a *reserved name* or *reserved word*. In general, any name that has special significance to the C compiler cannot be used as a variable name. Appendix A, "ANSI C Language Summary," provides a complete list of such reserved names.

You should always remember that upper- and lowercase letters are distinct in C. Therefore, the variable names `sum`, `Sum`, and `SUM` each refer to a different variable.

A variable name can be up to 31 characters in length.[1]

Some systems may permit names to be even longer than that. For example, UNIX Systems (as of System V Release 2) permit variable names to be as long as desired.

> **QUICK TIP**
>
> When deciding on the choice of a variable name, keep one recommendation in mind—don't be lazy. Pick names that reflect the intended use of the variable. The reasons are obvious. Just as with the comment statement, meaningful variable names can dramatically increase the readability of a program and will pay off in the debug and documentation phases. In fact, the documentation task will probably be greatly reduced since the program will be more self-explanatory.

Data Types and Constants

You have already been exposed to the C basic data type `int`. As you will recall, a variable declared to be of type `int` can be used to contain integral values only—that is, values that do not contain decimal places.

The C programming language provides the user with three other basic data types: `float`, `double`, and `char`. A variable declared to be of type `float` can be used for storing floating-point numbers (values containing decimal places). The `double` type is the same as type `float`, only with roughly twice the accuracy. Finally, the `char` data type can be used to store a single character, such as the letter "a," the digit character "6," or a semicolon; more on this later.

In C, any literal number, single character, or character string is known as a *constant*. For example, the number 58 represents a constant integer value. The character string `"Programming in C is fun.\n"` is an example of a constant character string. Expressions consisting entirely of constant values are called *constant expressions*. So the expression

```
128 + 7 - 17
```

[1] *See Chapter 15, "Working with Larger Programs," for a discussion on external names. Your system might not allow such names to be 31 characters long.*

is a constant expression because each of the terms of the expression is a constant value. But if `i` were declared to be an integer variable, then the expression

```
128 + 7 - i
```

would not represent a constant expression.

It is important to understand the concept of constants in order to fully understand data types and operations in C.

Type *int*

In C, an integer constant consists of a sequence of one or more digits. A minus sign preceding the sequence indicates that the value is negative. The values 158, -10, and 0 are all valid examples of integer constants. No embedded spaces are permitted between the digits, and values larger than 999 cannot be expressed using commas. (So the value 12,000 is not a valid integer constant and must be written as 12000.)

Two special formats in C enable integer constants to be expressed in a base other than decimal (base 10). If the first digit of the integer value is a zero, then the integer is taken as expressed in *octal* notation, that is, in base 8. In that case, the remaining digits of the value must be valid base-8 digits and therefore must be from 0 through 7. So, to express the value octal 50 in C, the notation `050` is used. Similarly, the C octal constant `0177` represents the decimal value 127 (1 x 64 + 7 x 8 + 7). An integer value can be displayed at the terminal in octal notation by using the format characters `%o` in the format string of a `printf` statement. In such a case, it is noted that the value will be displayed in octal *without* a leading zero. The format character `%#o` *will* cause a leading zero to be displayed before an octal value.

> **QUICK TIP**
>
> Don't forget that any integer value preceded by a `0` is considered to be an octal number by the compiler!

If an integer constant is preceded by a `0` and the letter `x` (either lowercase or uppercase), then the value is taken as being expressed in hexadecimal (base 16) notation. Immediately following the letter `x` are the digits of the hexadecimal value, which can be composed of the digits `0` through `9` and the letters `a` through `f` (or `A` through `F`). The letters represent the values 10 through 15,

respectively. So, to assign the hexadecimal value 5EB (= $16^2 \times 5 + 16 \times 14 + 11$ = 1,515 decimal) to an integer variable called `type_mask`, the statement

```
type_mask = 0x5EB;
```

can be used. To display an integer value in hexadecimal format at the terminal, use the format characters `%x`. So the statement

```
printf ("Value = %x\n", type_mask);
```

would display the value of `type_mask` in hexadecimal format, *without* the leading `0x`. To display the value with the leading `0x`, you would use the format characters `%#x`, as in

```
printf ("Value = %#x\n", type_mask);
```

Octal and hexadecimal constants frequently find their way into systems programs and more advanced programming applications. As such, this notation will not be used again until much later in this book.

Type *float*

A variable declared to be of type `float` can be used for storing values containing decimal places. A floating-point constant is distinguished by the presence of a decimal point. It is permissible to omit digits before the decimal point, or digits after the decimal point, but obviously it is not permissible to omit both. The values `3.`, `125.8`, and `-.0001` are all valid examples of floating-point constants. To display a floating-point value at the terminal, the `printf` conversion characters `%f` are used.

Floating-point constants can also be expressed in *scientific notation*. The value `1.7e-4` is a floating-point value expressed in this notation and represents the value 1.7×10^{-4}. The value before the letter `e` is known as the *mantissa*, while the value that follows is called the *exponent*. This exponent, which may be preceded by an optional plus or minus sign, represents the power of 10 that the mantissa is to be multiplied by. So, in the constant `2.25e-3`, the 2.25 is the value of the mantissa and -3 is the value of the exponent. This constant represents the value 2.25^{-3}, or 0.00225. Incidentally, the letter `e`, which separates the mantissa from the exponent, can be written in either lower- or uppercase.

In order to display a value in scientific notation, the format characters `%e` should be specified in the `printf` format string. The `printf` format characters `%g` can be used to let `printf` decide whether to display the floating-point value in normal floating-point notation or in scientific notation. This decision will be based upon the value of the exponent: if it's less than -4 or greater than 5, `%e` (scientific notation) format will be used; otherwise, `%f` format will be used.

QUICK TIP

Use the %g format characters for displaying floating-point numbers—it produces the most aesthetically pleasing output.

Type *double*

The type `double` is very similar to type `float`. It is used whenever the accuracy provided by a `float` variable is not sufficient. Variables declared to be of type `double` can store roughly twice as many significant digits as can a variable of type `float`. The precise number of digits that can be stored in either a `float` or `double` variable depends upon the particular computer system you are using. On the DEC VAXen, a `float` variable can contain approximately seven decimal digits, while a variable of type `double` has a precision of approximately 16 decimal digits.

Unless told otherwise, all floating-point constants are taken as `double` values by the C compiler. To explicitly express a `float` constant, append either an `f` or `F` to the end of the number, as in

```
12.5f
```

To display a `double` value at the terminal, the format characters `%f`, `%e`, or `%g`, which are the same format characters used to display a `float` value, can be used.

Type *char*

A `char` variable can be used to store a single character. A character constant is formed by enclosing the character within a pair of *single* quote marks. So `'a'`, `';'`, and `'0'` are all valid examples of character constants. The first constant represents the letter a, the second a semicolon, and the third the *character* zero—which is not the same as the *number* zero. Do not confuse a character constant, which is a single character enclosed in single quotes, with a character string, which is any number of characters enclosed in *double* quotes. We will learn more about this distinction in Chapter 10, "Character Strings."

The character constant `'\n'`—the newline character—is a valid character constant even though it seems to contradict the rule cited above. The reason for this is that the backslash character is a special character in the C system and does not actually count as a character. In other words, the C compiler treats the character `'\n'` as a single character, even though it is actually formed by two characters. In Chapter 10, you will find that the backslash character can be used for purposes other than just for advancing to the next line. The format characters `%c` can be used in a `printf` call to display the value of a `char` variable at the terminal.

In Program 4.1, the basic C data types are used. Notice how a value can be assigned to a variable at the time that the variable is declared.

Program 4.1

```
#include <stdio.h>
main ()
{
    int     integer_var = 100;
    float   floating_var = 331.79;
    double  double_var =  8.44e+11;
    char    char_var = 'W';

    printf ("integer_var = %i\n", integer_var);
    printf ("floating_var = %f\n", floating_var);
    printf ("double_var = %e\n", double_var);
    printf ("double_var = %g\n", double_var);
    printf ("char_var = %c\n", char_var);
}
```

Program 4.1 OUTPUT

```
integer_var = 100
floating_var = 331.789978
double_var = 8.440000e+11
double_var = 8.44e+11
char_var = W
```

The first statement of Program 4.1 declares the variable `integer_var` to be an integer variable and also assigns to it an initial value of 100, as if the following two statements had been used instead:

```
int  integer_var;
integer_var = 100;
```

In the second line of the program's output, you will notice that the value of 331.79 that we assigned to `floating_var` is actually displayed as 331.789978. In fact, the actual value that is displayed is quite dependent on the particular computer system you are using. The reason for this inaccuracy is the particular way that numbers are internally represented inside the computer system. You have probably come across the same type of inaccuracy when dealing with numbers on your pocket calculator. If you divide 1 by 3 on your calculator, you will get the result .33333333—with perhaps some additional 3s tacked on at the end. The string of 3s is the calculator's approximation of one-third. Theoretically, there should be an infinite number of 3s. But the calculator can only

hold so many digits, thus the inherent inaccuracy of the machine. The same type of inaccuracy applies to your computer system. Certain floating-point values cannot be exactly represented inside the computer's memory.

When displaying the values of `float` or `double` variables, you have the choice of three different formats. The `%f` characters are used to display values in a standard manner. Unless told otherwise, `printf` will always display a `float` or `double` value to six decimal places rounded. We will see later in this chapter how to select the number of decimal places that are displayed.

The `%e` characters are used to display the value of a `float` or `double` variable in scientific notation. Once again, six decimal places are automatically displayed by the system.

With the `%g` characters, `printf` chooses between `%f` and `%e` and also automatically removes from the display any trailing zeroes. If no digits follow the decimal point, it won't display that either.

In the last `printf` statement, the `%c` characters are used to display the single character `'W'` that we assigned to `char_var` when the variable was declared. Remember that while a character string (such as the first argument to `printf`) is enclosed within a pair of double quotes, a character constant must always be enclosed within a pair of single quotes.

Qualifiers *long*, *short*, *unsigned*, and *signed*

If the qualifier `long` is placed directly before the `int` declaration, the declared integer variable will be of extended accuracy on many computer systems. An example of a `long int` declaration might be

```
long int  factorial;
```

which would declare the variable `factorial` to be a `long` integer variable. As with `float`s and `double`s, the particular accuracy of a `long` variable depends upon the particular computer system. On the IBM PC, for example, the maximum positive value that can be stored inside a variable of type `int` is 32,767. Declaring a variable to be of type `long int` permits values up to 2^{31}-1 or 2,147,483,647 to be stored in the variable. On the SUN SPARC Station, an `int` and a `long int` both have the same accuracy (which is the same as that for a `long int` on the IBM PC) and can therefore both be used to store integer values up to 2,147,483,647.

A constant value of type `long int` is formed by optionally appending the letter `L` (upper- or lowercase) to the end of an integer constant. No spaces are permitted between the number and the `L`. So, the declaration

```
long int  number_of_points = 131071L;
```


declares the variable `number_of_points` to be of type `long int` with an initial value of 131,071.

To display the value of a `long int` at the terminal, the letter `l` is used as a modifier before the integer format characters `i`, `o`, or `x`. This means that the format characters `%li` can be used to display the value of a `long int` in decimal format, the characters `%lo` to display the value in octal format, and the characters `%lx` to display the value in hexadecimal format.

The `long` qualifier is also allowed in front of a `double` declaration, as in

```
long double  US_deficit_2000;
```

Whether or not a `long double` gives you more precision than a `double` on your machine is uncertain, so you'll have to check your documentation.

A `long double` constant is written as a floating constant with the letter `l` or `L` immediately following, as in

```
1.234e+7L
```

To display a `long double`, the L modifier is used. So `%Lf` would display a `long double` value in floating-point notation, `%Le` would display the same value in scientific notation, and `%Lg` would tell `printf` to choose between `%Lf` and `%Le`.

The qualifier `short`, when placed in front of the `int` declaration, tells the C compiler that the particular variable being declared will be used to store fairly small integer values. The motivation for using `short` variables is primarily one of conserving the computer's memory space, which may be an issue in cases where the program needs a lot of memory and the amount available is limited.

On some machines, a `short int` will take up half the amount of storage a regular `int` variable. On the VAX-11, a `short int` can be used to store integer values in the range -32,768 to 32,767. On the IBM PC, a `short int` and an `int` take up the same amount of space: 16 bits. In any case, you are guaranteed that the amount of space allocated for a `short int` will not be less than 16 bits.

There is no way to explicitly write a constant of type `short int` in C. To display a `short int` variable, place the letter `h` in front of any of the normal integer conversion characters: `%hi`, `%ho`, `%hx`.

The final qualifier that may be placed in front of an `int` variable is used when an integer variable will be used only to store positive numbers. The declaration

```
unsigned int  counter;
```

declares to the compiler that the variable `counter` will be used only to contain positive values. By restricting the use of an integer variable to the exclusive storage of positive integers, the accuracy of the integer variable is extended. On the IBM PC for example, a normal `int` variable can assume values from -32,768 through 32,767. The same variable declared to be of type `unsigned int` can assume values from 0 through 65,535.

An `unsigned int` constant is formed by placing the letter `u` (or `U`) after the constant, as in

```
0x00ffU
```

It's possible to combine the letters `u` (or `U`) and `l` (or `L`) when writing an integer constant, so

```
20000UL
```

tells the compiler to treat the constant 20000 as `unsigned long`.

An integer constant that's not followed by any of the letters `u`, `U`, `l`, or `L`, and that is too large to fit into a normal-sized `int`, is treated as an `unsigned int` by the compiler. If it's too small to fit into an `unsigned int`, the compiler treats it as a `long int`. Finally, if it still can't fit inside a `long int`, the compiler makes it an `unsigned long int`.

When declaring variables to be of type `long int`, `short int`, or `unsigned int`, it is permissable to omit the keyword `int`. So the `unsigned` variable `counter` could have been equivalently declared as

```
unsigned  counter;
```

You can also declare `char` variables to be `unsigned`. The reason for using such a declaration is discussed in Chapter 14, "More on Data Types."

The `signed` qualifier can be used to explicitly tell the compiler that a particular variable is a signed quantity. It's use is primarily in front of the `char` declaration, and further discussion on this is deferred until Chapter 14.

Don't worry if the discussions of these qualifiers seem a bit esoteric to you at this point. In later sections of this book, we will illustrate the use of many of these different types with actual program examples. Chapter 14 goes into more detail about data types and conversions.

Table 4.1 summarizes the basic data types and qualifiers.

Table 4.1. Basic data types.

Type	*Constant Examples*	*printf Chars*
`char`	`'a', '\n'`	`%c`
`short int`	-	`%hi, %hx, %ho`
`unsigned short int`	-	`%hu, %hx, %ho`
`int`	`12, -97, 0xFFE0, 0177`	`%i, %x, %o`
`unsigned int`	`12u, 100U, 0XFFu`	`%u, %x, %o`
`long int`	`12L, -2001, 0xffffL`	`%li, %lx, %lo`
`unsigned long int`	`12UL, 100ul, oxffeeUL`	`%lu, %lx, %lo`
`float`	`12.34f, 3.1e-5f`	`%f, %e, %g`
`double`	`12.34, 3.1e-5`	`%f, %e, %g`
`long double`	`12.341, 3.1e-5l`	`%Lf, %Le, %Lg`

Arithmetic Expressions

In C, just as in virtually all programming languages, the plus sign (`+`) is used to add two values; the minus sign (`-`) to subtract two values; the asterisk (`*`) to multiply two values; and the slash (`/`) to divide two values. These operators are known as *binary* arithmetic operators, since they operate on *two* values or terms.

We have seen how a simple operation like addition can be performed in C. The following program further illustrates the operations of subtraction, multiplication, and division. The last two operations performed in the program introduce the notion that one operator can have a higher priority, or *precedence*, over another operator. In fact, each operator in C has a precedence associated with it. This precedence is used to determine how an expression that has more than one operator is evaluated: The operator with the higher precedence is evaluated first. Expressions containing operators of the same precedence are either evaluated from left to right or from right to left, depending upon the operator. This is known as the *associative* property of an operator. Appendix A provides a complete list of operator precedences and their rules of association.

Program 4.2

```
/* Illustrate the use of various arithmetic operators */
#include <stdio.h>
main ()
{
    int  a = 100;
    int  b = 2;
    int  c = 25;
    int  d = 4;
    int  result;

    result = a - b;         /* subtraction */
    printf ("a - b = %i\n", result);

    result = b * c;         /* multiplication */
    printf ("b * c = %i\n", result);

    result = a / c;         /* division */
    printf ("a / c = %i\n", result);

    result = a + b * c;     /* precedence */
    printf ("a + b * c = %i\n", result);

    printf ("a * b + c * d =  %i\n", a * b + c * d);
}
```

Program 4.2 OUTPUT

```
a - b = 98
b * c = 50
a / c = 4
a + b * c = 150
a * b + c * d = 300
```

After declaring the integer variables `a`, `b`, `c`, `d`, and `result`, the program assigns the result of subtracting `b` from `a` to `result` and then proceeds to display its value with an appropriate `printf` call.

The next statement

```
result = b * c;
```

has the effect of multiplying the value of `b` by the value of `c` and storing the product in `result`. The result of the multiplication is then displayed using a `printf` call that should be very familiar to you by now.

The next program statement introduces the division operator—the slash. The result of 4 as obtained by dividing 100 by 25 is displayed by the `printf` statement immediately following the division of `a` by `c`.

On most computer systems, attempting to divide a number by zero will result in abnormal termination of the program. Even if the program does not terminate abnormally, the results obtained by such a division will be meaningless. In Chapter 6, "Making Decisions," we will see how we can check for division by zero before the division operation is performed. If it is determined that the divisor is zero, then an appropriate action (such as displaying a message at the terminal) can be taken and the division operation averted.

QUICK TIP

Make sure you get into the habit of ensuring that any division in your program will not be by zero. This will prevent your programs from terminating abnormally.

The expression

```
a + b * c
```

does not produce the result of 2550 (102 * 25) as might be expected; rather, the result as displayed by the corresponding `printf` statement is shown as 150. This is because C, like most other programming languages, has rules for the order of evaluating multiple operations or terms in an expression. Evaluation of an expression generally proceeds from left to right. However, the operations of multiplication and division are given precedence over the operations of addition and subtraction. Therefore, the expression

```
a + b * c
```

is evaluated as

```
a + (b * c)
```

by the C system. (This is the same way that this expression would be evaluated if we were to apply the basic rules of algebra.)

If we wish to alter the order of evaluation of terms inside an expression, then parentheses can be used. In fact, the last expression listed above is a perfectly valid C expression. Thus the statement

```
result = a + (b * c);
```

could have been substituted in Program 4.2 to achieve identical results. However, if the expression

```
result = (a + b) * c;
```

were used instead, then the value assigned to result would be 2500, since the value of a (100) would be added to the value of b (2) before multiplication by the value of c (25) would take place. Parentheses may be nested, and evaluation of the expression will proceed outward from the innermost set of parentheses. Just be sure to have as many closed parentheses as there are open ones.

You will notice from the last statement in Program 4.2 that it is perfectly valid to give an expression as an argument to printf without having to first assign the result of the expression evaluation to a variable. The expression

```
a * b + c * b
```

is evaluated according to the rules stated above as

```
(a * b) + (c * b)
```

or

```
(100 * 2) + (25 * 4)
```

and the result of 300 "handed" to the printf routine.

Integer Arithmetic and the Unary Minus Operator

Program 4.3 reinforces what we have just discussed and introduces the concept of integer arithmetic.

Program 4.3

```
/* More arithmetic expressions */
#include <stdio.h>
main ()
{
    int a = 25;
    int b = 2;
    int result;
    printf ("a = %d\n", a);
    printf ("b = %d\n\n", b);

    result = a / b * b;
    printf ("a / b * b = %d\n", result);
    result = a / b * b + a % b;
    printf ("a / b * b + a %% b = %d\n", result);
    result = a / (b * b) + a % b;
    printf ("a / (b * b) + a %% b = %d\n", result);
    result = a / b * b + a % b * b;
    printf ("a / b * b + a %% b * b = %d\n", result);
}
```


Program 4.3 OUTPUT

```
a = 25
b = 2

a / b * b = 24
a / b * b + a % b = 25
a / (b * b) + a % b = 25
a / b * b + a % b * b = 25
```

We inserted extra blank spaces between int and the declaration of a, b, and result in the first three statements to align the declaration of each variable. This helps to make the program more readable. You also may have noticed in each program presented so far that a blank space was placed around each operator. This too is not required and is done solely for aesthetic reasons. In general, you may add extra blank spaces just about anywhere that a single blank space is allowed. A few extra presses of the spacebar will prove worthwhile if the resulting program is easier to read.

The expression in the first printf call of Program 4.3 reinforces the notion of operator precedence. Evaluation of this expression proceeds as follows:

1. Since division has higher precedence than addition, the value of a (25) is divided by 5 first. This gives the intermediate result of 5.
2. Since multiplication also has higher precedence than addition, the intermediate result of 5 is next multiplied by 2, the value of b, giving a new intermediate result of 10.
3. Finally, the addition of 6 and 10 is performed, giving a final result of 16.

The second printf statement introduces a new twist. We would expect that dividing a by b and then multiplying by b would return to us the value of a, which we have set to 25. But this does not seem to be the case, as shown by the output display of 24. Did the computer lose a bit somewhere along the way? Very unlikely. The fact of the matter is that this expression was evaluated using integer arithmetic.

If you glance back at the declarations for the variables a and b, you will recall that they were both declared to be of type int. Now, whenever a term to be evaluated in an expression consists of two integers, the C system performs the operation using integer arithmetic. In such a case, all decimal portions of numbers are lost. Therefore, when the value of a is divided by the value of b, or 25 is divided by 2, we get an intermediate result of 12 and not 12.5 as you might expect. Multiplying this intermediate result by 2 gives us the final result of 24, thus explaining the "lost" digit.

QUICK TIP

Don't forget that if you divide two integers, you always get an integer result.

As can be seen from the next-to-last `printf` statement in Program 4.3, if we perform the same operation using floating-point values instead of integers, we obtain the expected result.

The decision of whether to use a `float` variable or an `int` variable should be made based upon the variable's intended use. If you don't need any decimal places, then use an integer variable. The resulting program will be more *efficient*—that is, it will execute faster on most computers. On the other hand, if you need the decimal place accuracy, then the choice is clear. The only question that you then must decide is whether to use a `float` or `double`. The answer to this question will depend upon the desired accuracy of the numbers you are dealing with, as well as their magnitude.

In the last `printf` statement, the value of the variable a is *negated* by use of the *unary minus* operator. A unary operator is one which operates on a single value, as opposed to a binary operator, which operates on two values. The minus sign actually has a dual role: As a binary operator, it is used for subtracting two values. As a unary operator, it is used to negate a value.

The unary minus operator has higher precedence than all other arithmetic operators, except for the unary plus operator (+), which has the same precedence. So the expression

```
c = -a * b;
```

will result in the multiplication of `-a` by `b`. Once again, in Appendix A you will find a table summarizing the various operators and their precedences.

The Modulus Operator

The last operator to be presented in this chapter is the *modulus* operator, which is symbolized by the percent sign (%). Try to determine how this operator works by analyzing the output from Program 4.4.

Program 4.4

```
/* The modulus operator */
#include <stdio.h>
main ()
{
    int  a = 25, b = 5, c = 10, d = 7;

    printf ("a %% b = %i\n", a % b);
    printf ("a %% c = %i\n", a % c);
    printf ("a %% d = %i\n", a % d);
    printf ("a / d * d + a %% d = %i\n",
             a / d * d + a % d);
}
```


Program 4.4 OUTPUT

```
a % b = 0
a % c = 5
a % d = 4
a / d * d + a % d = 25
```

The first statement inside `main` declares and initializes the variables `a`, `b`, `c`, and `d` in a single statement.

In the first `printf` call, you will notice that we used *two* percent signs in the format string; yet, if you examine the program's output, you will notice that only a single percent sign was printed. The reason for this is the special significance of the `%` sign to the `printf` routine. As you know, `printf` uses the character that immediately follows the percent sign to determine how to print the next argument. However, if it is another percent sign that follows, then the `printf` routine takes this as an indication that you really intend to display a percent sign and inserts one at the appropriate place in the program's output.

You are correct if you concluded that the function of the modulus operator `%` is to give the remainder of the first value divided by the second value. In the first example, the remainder after 25 is divided by 5 is displayed as 0. If we divide 25 by 10, we get a remainder of 5, as verified by the second line of output. Dividing 25 by 7 gives a remainder of 4, as shown in the third output line.

The last line of output in Program 4.4 requires a bit of explanation. First, you will notice that the program statement has been written on two lines. This is perfectly valid in C. In fact, a program statement may be continued to the next line at any point where a blank space could be used. (An exception to this occurs when dealing with character strings—a topic discussed in Chapter 10.) At times it may not only be desirable, but perhaps even necessary to continue a program statement onto the next line. The continuation of the `printf` call in Program 4.4 is indented to visually show that it is a continuation of the preceding program statement.

Let us now turn our attention to the expression that is evaluated in this last statement. You will recall that any operations between two integer values in C are performed with integer arithmetic. Therefore, any remainder resulting from the division of two integer values will simply be discarded. Dividing 25 by 7, as indicated by the expression `a / d`, gives an intermediate result of 3. Multiplying this value by the value of `d`, which is 7, produces the intermediate result of 21. Finally, adding in the remainder of dividing `a` by `d`, as indicated by the expression `a % d`, leads to the final result of 25. It is no coincidence that this value is the same as the value of the variable `a`. In general, the expression

```
a / b * b + a % b
```

will always equal the value of `a`, assuming of course that `a` and `b` are both integer values. In fact, the modulus operator `%` is defined to work only with integer values.

As far as precedence is concerned, the modulus operator has equal precedence to the multiplication and division operators. This implies, of course, that an expression such as

```
table + value % TABLE_SIZE
```

will be evaluated as

```
table + (value % TABLE_SIZE)
```

Integer and Floating-Point Conversions

In order to effectively develop C programs, you must understand the rules that are used for the implicit conversion of floating-point and integer values in C. Program 4.5 demonstrates some of the simple conversions between numeric data types. In Chapter 14, "More on Data Types," the rules that are followed for conversion between other data types are described in detail.

Program 4.5

```
/* Basic conversions in C */
#include <stdio.h>
main ()
{
    float  f1 = 123.125, f2;
    int    i1, i2 = -150;
    char   c = 'a';

    i1 = f1;                /* floating to integer conversion */
    printf ("%f assigned to an int produces %i\n", f1, i1);

    f1 = i2;                /* integer to floating conversion */
    printf ("%i assigned to a float produces %f\n", i2, f1);

    f1 = i2 / 100;          /* integer divided by integer */
    printf ("%i divided by 100 produces %f\n", i2, f1);

    f2 = i2 / 100.0;        /* integer divided by a float */
    printf ("%i divided by 100.0 produces %f\n", i2, f2);
}
```


Program 4.5 OUTPUT

```
123.125000 assigned to an int produces 123
-150 assigned to a float produces -150.000000
-150 divided by 100 produces -1.000000
-150 divided by 100.0 produces -1.500000
```

Whenever a floating-point value is assigned to an integer variable in C, the decimal portion of the number gets *truncated*. So when the value of `f1` is assigned to `i1` in the above program, the number 123.125 gets truncated, which means that only its integer portion, or 123, is stored into `i1`. The first line of the program's output verifies that this is the case.

Assigning an integer variable to a floating variable does not cause any change in the value of the number. The value is simply converted by the system and stored into the floating variable. The second line of the program's output verifies that the value of `i2`, -150, was correctly converted and stored into the `float` variable `f1`.

The next two lines of the program's output illustrate two points that must be remembered when forming arithmetic expressions. The first has to do with integer arithmetic, which we have already discussed in this chapter. Whenever two operands in an expression are integers (and this applies to `short`, `unsigned`, and `long` integers as well), the operation is carried out under the rules of integer arithmetic. Therefore, any decimal portion resulting from a division operation will be discarded, even if the result is assigned to a floating variable (as we did in the program). So when the integer variable `i2` is divided by the integer constant 100, the system performs the division as an integer division. The result of dividing -150 by 100, which is -1, is therefore the value that is stored into the `float` variable `f1`.

The next division that is performed in the above program involves an integer variable and a floating-point constant. Any operation between two values in

C will be performed as a floating-point operation if *either* value is a floating-point variable or constant. Therefore, when the value of `i2` is divided by 100.0, the system treats the division as a floating-point division and produces the result of -1.5, which is assigned to the `float` variable `f1`.

We have reached the end of discussion on variables, data types, and expressions. If you are familiar with other programming languages, you may have noticed that there is no operator for raising a number to a power (exponentiation). Luckily, there is a function called `pow` in the program library for performing exponentiation.

Try your hand at the exercises presented on the following pages before proceeding to the next chapter, which introduces the concept of program looping.

Exercises

1. If you have access to a computer facility that supports the C programming language, type in and run the five programs presented in this chapter. Compare the output produced by each program with the output presented after each program.

2. Which of the following are invalid variable names? Why?

`Int`	`char`	`6_05`
`calloc`	`Xx`	`alpha_beta_routine`
`floating`	`_1312`	`z`
`ReInitialize`	`_`	`A$`

3. Which of the following are invalid constants? Why?

`123.456`	`0x10.5`	`0X0G1`
`0001`	`0xFFFF`	`123L`
`0Xab05`	`0L`	`-597.25`
`123.5e2`	`.0001`	`+12`
`98.6F`	`98.7U`	`17777s`
`0996`	`-12E-12`	`07777`
`1234uL`	`1.2Fe-7`	`15,000`
`1.234L`	`197u`	`100U`
`0XABCDEFL`	`0xabcu`	`+123`

4. Write a program that converts 27° from degrees Fahrenheit (*F*) to degrees Celsius (*c*) using the formula

$C = (F - 32)/1.8$

5. What output would you expect from the following program?

```
main ()
{
   char  c, d;

   c = 'd';
   d = c;
   printf ("d = %c\n", d);
}
```

6. Write a program to evaluate the polynomial

$3x^3 - 5x^2 + 6$

for $x = 2.55$.

7. Write a program that evaluates the following expression and displays the results. (Remember to use exponential format to display the result.)

$(3.31 \times 10^{-8} x + 2.01 \times 10^{-7}) / (7.16 \times 10^{-6} + 2.01 \times 10^{-8})$

8. To round off an integer *i* to the next largest even multiple of another integer *j*, the following formula can be used:

```
Next_multiple = i + j - i % j
```

For example, to round off 256 days to the next largest number of days evenly divisible by a week, values of $i = 256$ and $j = 7$ can be substituted into the above formula as follows:

```
Next_multiple     = 256 + 7 - 256 % 7
                  = 256 + 7 - 4
                  = 259
```

Write a program to find the next largest even multiple for the following values of *i* and *j*:

i	*j*
365	7
12,258	23
996	4

5

CHAPTER

Program Looping

[illegible]

```
    *
   * *
  * * *
 * * * *
* * * * *
```

[illegible]

[illegible]

The technique of Program 5.1 works fine for calculating relatively small triangular numbers. But what would happen if we needed to find the value of the 200th triangular number, for example? It certainly would be a bit tedious to have to modify Program 5.1 to explicitly add up all of the integers from 1 to 200. Luckily, there is an easier way.

Program 5.1

```
/* Program to calculate the eighth triangular number */
#include <stdio.h>
main ()
{
        int  triangular_number;

        triangular_number = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;

        printf ("The eighth triangular number is %i\n",
                        triangular_number);
}
```

Program 5.1 OUTPUT

```
The eighth triangular number is 36
```

One of the fundamental properties of a computer is its ability to repetitively execute a set of statements. These *looping* capabilities enable programmers to develop concise programs containing repetitive processes that could otherwise require thousands or even millions of program statements to perform. The C programming language contains three different program statements for program looping. They are known as the `for` statement, the `while` statement, and the `do` statement. Each of these statements will be described in detail in this chapter.

The *for* Statement

Why don't we dive right in and take a look at a program that uses the `for` statement? The purpose of Program 5.2 is to calculate the 200th triangular number. See if you can determine how the `for` statement works.

Some explanation is owed for Program 5.2. The method employed to calculate the 200th triangular number is really the same as that used to calculate the 8th triangular number in the previous program—the integers from 1 to 200 are

summed. The `for` statement provides the mechanism that enables us to avoid having to explicitly write out each integer from 1 to 200. In a sense, this statement is used to "generate" these numbers for us.

Program 5.2

```
/* Program to calculate the 200th triangular number */
/* Introduction of the for statement                */
#include <stdio.h>
main ()
{
    int  n, triangular_number;

    triangular_number = 0;

    for ( n = 1;  n <= 200;  n = n + 1 )
    triangular_number = triangular_number + n;

    printf ("The 200th triangular_number is %i\n",
                        triangular_number);
}
```

Program 5.2 OUTPUT

```
The 200th triangular number is 20100
```

The general format of the `for` statement is as follows:

```
for ( init_expression; loop_condition; loop_expression )
    program statement ;
```

The three expressions that are enclosed within the parentheses—*init_expression*, *loop_condition*, and *loop_expression*—set up the environment for the program loop. The program statement that immediately follows (which is of course terminated by a semicolon) can be any valid C program statement and constitutes the *body of the loop*. This statement will be executed as many times as specified by the parameters set up in the `for` statement.

The first component of the `for`, labeled *init_expression*, is used to set the initial values *before* the loop begins. In Program 5.2, this portion of the `for` statement is used to set the initial value of `n` to 1. As you can see, an assignment is a valid form of an expression.

The second component of the `for` statement specifies the condition or conditions that are necessary *in order for the loop to continue*. In other words, looping continues *as long as* this condition is satisfied. Once again referring to Program 5.2, we find that the *`loop_condition`* of the `for` is specified by the *relational expression*

```
n <= 200
```

This expression can be read as "`n` less than or equal to 200." The "less than or equal to" operator (which is the less than character < followed immediately by the equals sign =) is only one of several relational operators provided in the C programming language. These operators are used to test specific conditions. The answer to the test will be "yes" or, more commonly, TRUE if the condition is satisfied and "no" or FALSE if the condition is not satisfied.

Table 5.1 lists all the relational operators that are available in C.

Table 5.1. Relational operators.

Operator	*Meaning*	*Example*
`==`	Equal to	`count == 10`
`!=`	Not equal to	`flag != DONE`
`<`	Less than	`a < b`
`<=`	Less than or equal to	`low <= high`
`>`	Greater than	`pointer > end_of_list`
`>=`	Greater than or equal to	`j >= 0`

The relational operators have lower precedence than all arithmetic operators. This means, for example, that an expression such as a

```
a < b + c
```

will be evaluated as

```
a < (b + c)
```

as you would expect. It would be TRUE if the value of a were less than the value of `b + c` and FALSE otherwise.

Pay particular attention to the "is equal to" operator == and do not confuse its use with the assignment operator =. The expression

```
a == 2
```


tests if the value of a is equal to 2, whereas the expression

```
a = 2
```

assigns the value 2 to the variable a.

The choice of which relational operator to use will obviously depend on the particular test being made and in some instances on your particular preferences. For example, the relational expression

```
n <= 200
```

can be equivalently expressed as

```
n < 201
```

Returning to our example, the program statement that forms the body of the `for` loop

```
triangular_number = triangular_number + n;
```

will be repetitively executed *as long as the result of the relational test is TRUE*, or in this case, as long as the value of `n` is less than or equal to 200. This program statement has the effect of adding the value of `triangular_number` to the value of `n` and storing the result back into the value of `triangular_number`.

When the *`loop_condition`* is no longer satisfied, execution of the program will continue with the program statement immediately following the for loop. In our program, execution will continue with the printf statement once the loop has terminated.

The final component of the for statement contains an expression that is evaluated each time *after* the body of the loop is executed. In Program 5.2, this *`loop_expression`* adds 1 to the value of n. Therefore, the value of n will be incremented by 1 each time after its value has been added into the value of triangular_number and will range in value from 1 through 201.

It is worth noting that the last value that `n` attains, namely 201, is *not* added into the value of `triangular_number` since the loop is terminated *as soon as* the looping condition is no longer satisfied, or as soon as `n` equals 201.

In summary then, execution of the `for` statement proceeds as follows:

1. The initial expression is evaluated first. This expression usually sets a variable that will be used inside the loop, generally referred to as an *index* variable, to some initial value such as 0 or 1.

2. The looping condition is evaluated. If the condition is not satisfied (the expression is FALSE), then the loop is immediately terminated. Execution continues with the program statement that immediately follows the loop.

3. The program statement that constitutes the body of the loop is executed.
4. The looping expression is evaluated. This expression is generally used to change the value of the index variable, frequently by adding 1 to it or subtracting 1 from it.
5. Return to Step 2.

QUICK TIP

Remember that the looping condition is evaluated immediately on entry into the loop, before the body of the loop has even executed one time. Also, remember not to put a semicolon after the close parenthesis at the end of the loop (this will immediately end the loop).

Since Program 5.2 actually generates all of the first 200 triangular numbers on its way to its final goal, it might be nice to generate a table of these numbers. To save space, however, we will assume that we just want to print a table of the first 10 triangular numbers. Program 5.3 below performs precisely this task!

Program 5.3

```
/* Program to generate a table of triangular numbers */
#include <stdio.h>
main ()
{
    int  n, triangular_number;

    printf ("TABLE OF TRIANGULAR NUMBERS\n\n");
    printf (" n     Sum from 1 to n\n");
    printf ("---    --------------\n");

    triangular_number = 0;

    for ( n = 1;  n <= 10;  ++n )
    {
    triangular_number = triangular_number + n;
    printf (" %i              %i\n", n, triangular_number);
    }
}
```


Program 5.3 OUTPUT

```
TABLE OF TRIANGULAR NUMBERS
 n     Sum from 1 to n
---    --------------
1               1
2               3
3               6
4               10
5               15
6               21
7               28
8               36
9               45
10               55
```

It is always a good idea to add some extra `printf` statements to a program in order to provide more meaning to the output. In Program 5.3, the purpose of the first three `printf` statements is simply to provide a general heading and to label the columns of the output. You will notice that the first `printf` statement contains two newline characters. As you would expect, this has the effect of not only advancing to the next line but also inserting an extra blank line into the display.

After the appropriate headings have been displayed, the program proceeds to calculate the first 10 triangular numbers. The variable `n` is used to count the current number whose "sum from 1 to n" we are computing, while the variable `triangular_number` is used to store the value of triangular number `n`.

Execution of the `for` statement commences by setting the value of the variable `n` to 1. You will recall that we mentioned earlier that the program statement immediately following the `for` statement constitutes the body of the program loop. But what happens if we wish to repetitively execute not just a single program statement but a group of program statements? This can be accomplished by enclosing all such program statements within a pair of braces. The system will then treat this group or *block* of statements as a single entity. In general, any place in a C program that a single statement is permitted, a block of statements can be used, provided that you remember to enclose the block within a pair of braces.

Therefore, in Program 5.3, both the expression that adds `n` into the value of `triangular_number` and the `printf` statement that immediately follows constitute the body of the program loop. Pay particular attention to the way the program statements are indented. At a quick glance, it is easy to determine which statements form part of the `for` loop. You should also note that programmers use different coding styles. Some prefer to type the loop this way:

```
for ( n = 1;  n <= 10;  ++n ) {
    triangular_number = triangular_number + n;
    printf ("%i                %i\n", n, triangular_number);
}
```

Here the opening brace is placed on the same line as the `for`. This is strictly a matter of taste and has no effect on the program.

The next triangular number is calculated by simply adding the value of `n` to the previous triangular number. The first time through the `for` loop, the "previous" triangular number is zero, so the new value of `triangular_number` when `n` is equal to 1 is simply the value of `n`, or 1. The values of `n` and `triangular_number` are then displayed, with an appropriate number of blank spaces inserted in the format string to ensure that the values of the two variables line up under the appropriate column headings.

Since the body of the loop has now been executed, the looping expression is evaluated next. The expression in this `for` statement appears a bit strange, however. Surely we must have made a typographical mistake and meant to insert the expression

```
n = n + 1
```

instead of the funny-looking expression

```
++n
```

The fact of the matter is that `++n` is actually a perfectly valid C expression. It introduces us to a new (and rather unique) operator in the C programming language—the *increment operator*. The function of the double plus sign—or the increment operator—is to add 1 to its operand. Since addition by 1 is such a common operation in programs, a special operator was created solely for this purpose. Therefore, the expression `++n` is equivalent to the expression `n = n + 1`. While at first glance it might appear that `n = n + 1` is more readable, you will very soon get used to the function of this operator and will even learn to appreciate its succinctness.

Of course, no programming language that offered an increment operator to add 1 would be complete without a corresponding operator to subtract 1. And as you would guess, the name of this operator is the *decrement operator* and is symbolized by the double minus sign. So, an expression in C that reads

```
bean_counter = bean_counter - 1
```

can be equivalently expressed using the decrement operator as

```
--bean_counter
```

Some programmers prefer to put the `++` or `--` after the variable name, as in `n++` or `bean_counter--`. This is acceptable, and is a matter of personal preference.

QUICK TIP

One slightly disturbing thing that you may have noticed in Program 5.3's output is the fact that the 10th triangular number does not quite line up under the previous triangular numbers. This is because the number 10 takes up two print positions, whereas the previous values of n, 1 through 9, took up only one print position. Therefore, the value 55 is effectively "pushed over" one extra position in the display.

This minor annoyance can be corrected if we substitute the following `printf` statement in place of the corresponding statement from Program 5.3.

```
printf ("%2i                %i\n", n, triangular_number);
```

To verify that this change will do the trick, here is the output from the modified program (we'll call it Program 5.3A).

Program 5.3A OUTPUT

```
n           Sum from 1 to n
--          ---------------
 1                  1
 2                  3
 3                  6
 4                 10
 5                 15
 6                 21
 7                 28
 8                 36
 9                 45
10                 55
```

The primary change made to the `printf` statement was the inclusion of a *field width specification*. The characters `%2i` tell the `printf` routine that not only do we wish to display the value of an integer at that particular point, but also that the size of the integer to be displayed should take up two columns in the display. Any integer that would normally take up less than two columns (that is, the integers 0 through 9) will be displayed with a leading space. This is known as *right justification*.

Thus, by using a field width specification of `%2i` we guarantee that at least two columns will be used for displaying the value of `n` and, therefore, ensure that the values of `triangular_number` will be lined up.

If the value that is to be displayed requires more columns than are specified by the field width, `printf` simply ignores the field width specification and uses as many columns as are necessary to display the value.

Field width specifications can be used for displaying values other than integers as well. We will see some examples of this in programs that will be coming up shortly.

Program Input

Program 5.2 calculates the 200th triangular number—and nothing more. What if we wanted to calculate the 50th or the 100th triangular number instead? Well, if that were the case, then we would have to go back and change the program so that the `for` loop would be executed the correct number of times. We would also have to change the `printf` statement to display the correct message.

An easier solution might be if we could somehow have the program ask us which triangular number we wished to calculate. Then, once we had given our answer, the program could calculate the desired triangular number for us. Such a solution can be effected in C by use of a routine called `scanf`. The `scanf` routine is very similar in concept to the `printf` routine. Whereas the latter routine is used to display values at the terminal, the function of the former routine is to enable the programmer to type values *into* the program. Program 5.4 asks the user which triangular number should be calculated, proceeds to calculate that number, and then displays the results.

Program 5.4

```
#include <stdio.h>
main ()
{
    int  n, number, triangular_number;

    printf ("What triangular number do you want? ");
    scanf  ("%i", &number);

    triangular_number = 0;
    for ( n = 1;  n <= number;  ++n )
    triangular_number = triangular_number + n;

    printf ("Triangular number %i is %i\n", number,
    triangular_number);
}
```


In the program output that follows, the number typed in by the user (100) is set in bolder type to distinguish it from the output displayed by the program.

Program 5.4 OUTPUT

```
What triangular number do you want? 100
Triangular number 100 is 5050
```

According to the output, the number 100 was typed in by the user. The program then proceeded to calculate the 100th triangular number and displayed the result of 5050 at the terminal. The user could have just as easily typed in the number 10, or 30, if he or she desired to calculate those particular triangular numbers.

The first `printf` statement in Program 5.4 is used to prompt the user to type in a number. Of course, it is always nice to remind the user what it is you want entered. After the message is printed, the `scanf` routine is called. The first argument to `scanf` is the format string and is very similar to the format string used by `printf`. In this case, the format string doesn't tell the system what types of values are to be displayed but rather what types of values are to be read in from the terminal. Like `printf`, the `%i` characters are used to specify an integer value.

The second argument to the `scanf` routine specifies *where* the value that is typed in by the user is to be stored. The `&` character before the variable `number` is necessary in this case. Don't worry about its function here, though. We will discuss this character, which is actually an operator, in great detail in Chapter 11, "Pointers."

QUICK TIP

Don't ever forget to put the leading `&` in front of the variable name in the `scanf` function call. If you do, it will cause strange results and may cause your program to terminate abnormally.

Given the preceding discussion, we can now see that the `scanf` call from Program 5.4 specifies that an integer value is to be read from the terminal and stored into the variable `number`. This value represents the particular triangular number that the user wishes to have calculated.

Once this number has been typed in (and the "Return" key on the keyboard pressed to signal that typing of the number is completed), the program then proceeds to calculate the requested triangular number. This is done the same way as in Program 5.2, the only difference being that instead of using 200 as the limit, `number` is used.

After the desired triangular number has been calculated, the results are displayed, and execution of the program is then complete.

Nested *for* Loops

Program 5.4 gives the user the flexibility to have the program calculate any triangular number that is desired. But suppose the user had a list of five triangular numbers to be calculated? Well, in such a case, the user could simply execute the program five times, each time typing in the next triangular number from the list to be calculated.

Another way to accomplish the same goals, and a far more interesting method as far as learning about C is concerned, is to have the program handle the situation. This can best be accomplished by inserting a loop into the program to simply repeat the entire series of calculations five times. We know by now that the `for` statement can be used to set up such a loop. The following program and its associated output illustrate this technique.

Program 5.5

```
#include <stdio.h>
main ()
{
     int  n, number, triangular_number, counter;

     for ( counter = 1;  counter <= 5;  ++counter )
     {
          printf ("What triangular number do you want? ");
          scanf  ("%i", &number);

          triangular_number = 0;

          for ( n = 1;  n <= number;  ++n )
               triangular_number = triangular_number + n;

          printf ("Triangular number %i is %i\n\n",
          number, triangular_number);
     }
}
```


Program 5.5 OUTPUT

```
What triangular number do you want? 12
Triangular number 12 is 78

What triangular number do you want? 25
Triangular number 25 is 325

What triangular number do you want? 50
Triangular number 50 is 1275

What triangular number do you want? 75
Triangular number 75 is 2850

What triangular number do you want? 83
Triangular number 83 is 3486
```

The program consists of two levels of `for` statements. The outermost `for` statement

```
for ( counter = 1;  counter <= 5;  ++counter )
```

specifies that the program loop is to be executed precisely five times. This can be seen because the value of `counter` will be initially set to 1 and will be incremented by 1 *until* it is no longer less than or equal to 5 (in other words, until it reaches 6).

Unlike the previous program examples, the variable counter is not used anywhere else within the program. Its function is solely as a loop counter in the `for` statement. Nevertheless, since it *is* a variable, it must be declared in the program.

The program loop actually consists of all the remaining program statements, as indicated by the braces. It might be easier for you to comprehend the way this program operates if you conceptualize it as follows:

```
For 5 times
{
     Get the number from the user.

     Calculate the requested triangular number.

     Display the results.
}
```

The portion of the loop referred to in the preceding as *Calculate the requested triangular number* actually consists of setting the value of the variable `triangular_number` to 0 *plus the* `for` *loop that calculates the triangular number.*

Thus we see that we have a `for` statement that is actually contained *within* another `for` statement. This is perfectly valid in C, and nesting may continue even further to any desired level.

The proper use of indentation becomes even more critical when dealing with more sophisticated program constructs, such as nested `for` statements. At a quick glance you can easily determine which statements are contained within each `for` statement. (To see how unreadable a program can be if correct attention isn't paid to formatting, see Exercise 5 at the end of this chapter.)

for Loop Variants

Before leaving this discussion of the `for` loop, we should mention some of the syntactic variations that are permitted in forming this loop. It may very well be that when writing a `for` loop that you will discover that you have more than one variable that you would like to initialize before the loop begins, or perhaps more than one expression that you would like evaluated each time through the loop. We can include multiple expressions in any of the fields of the `for` loop, provided that we separate such expressions by commas. For example, in the `for` statement that begins

```
for ( i = 0, j = 0;  i < 10;  ++i )
    ...
```

the value of `i` is set to zero *and* the value of `j` is set to zero before the loop begins. The two expressions `i = 0` and `j = 0` are separated from each other by a comma, and both expressions are considered part of the *`init_expression`* field of the loop. As another example, the `for` loop that starts

```
for ( i = 0, j = 100;  i < 10;  ++i, j = j - 10 )
    ...
```

sets up two index variables, `i` and `j`; the former initialized to 0 and the latter to 100 before the loop begins. Each time after the body of the loop is executed, the value of `i` will be incremented by 1, while the value of `j` will be decremented by 10.

Just as the need may arise to include more than one expression in a particular field of the `for` statement, the need may arise to *omit* one or more fields from the statement. This can be done simply by omitting the desired field and marking its place with a semicolon. The most common application for the omission of a field in the `for` statement occurs when there is no initial expression that needs to be evaluated. The *`init_expression`* field can simply be "left blank" in such a case, as long as the semicolon is still included:

```
for (  ;  j != 100;  ++j )
    ...
```


This statement might be used if `j` were already set to some initial value before the loop was entered.

A `for` loop that has its *`looping_condition`* field omitted effectively sets up an infinite loop; that is, a loop that theoretically will be executed forever. Such a loop can be used provided there is some other means used to exit from the loop (such as executing a `return`, `break`, or `goto` statement as discussed later in this book).

The *while* Statement

The `while` statement further extends the C language's repertoire of looping capabilities. The syntax of this frequently used construct is simply

```
while ( expression )
        program statement
```

The *`expression`* specified inside the parentheses is evaluated. If the result of *`expression`* evaluation is TRUE, then *`program statement`* that immediately follows is executed. After execution of this statement (or statements if enclosed in braces), *`expression`* is once again evaluated. If the result of the evaluation is TRUE, then *`program statement`* is once again executed. This process continues until *`expression`* finally evaluates FALSE, at which point the loop is terminated. Execution of the program then continues with the statement that follows *`program statement`*.

As an example of its use, the following program sets up a `while` loop, which merely counts from 1 to 5.

Program 5.6

```
/* This program introduces the while statement */
#include <stdio.h>
main ()
{
    int  count = 1;

    while ( count <= 5 )
    {
        printf ("%i\n", count);
        ++count;
    }
}
```

Program 5.6 OUTPUT

```
1
2
3
4
5
```

The program initially sets the value of `count` to 1. Execution of the `while` loop then begins. Since the value of `count` is less than or equal to 5, the statement that immediately follows is executed. The braces serve to define both the `printf` statement and the statement that increments `count` as the body of the `while` loop. From the output of the program, we can readily observe that this loop is executed precisely 5 times, or until the value of `count` reaches 6.

You may have realized from this program that we could have readily accomplished the same task by using a `for` statement. In fact, a `for` statement can always be translated into an equivalent `while` statement, and vice versa. For example, the general `for` statement

```
for ( init_expression;  loop_condition;  loop_expression ) program
statement
```

can be equivalently expressed in the form of a `while` statement as

```
init_expression;
while ( loop_condition )
{
        program statement
        loop_expression;
}
```

Once you become familiar with the use of the `while` statement, you will gain a better feel as to when it seems more logical to use a `while` and when to use a `for`.

QUICK TIP

In general, a loop executed a predetermined number of times is a prime candidate for implementation as a `for` statement. Also, if the initial expression, looping expression, and looping condition all involve the same variable, then the `for` statement is probably the right choice.

The next program provides another example of the use of the `while` statement. The program computes the *greatest common divisor* of two integer values. The greatest common divisor (we'll abbreviate it hereafter as *gcd*) of two integers is the largest integer value that evenly divides the two integers. For example, the *gcd* of 10 and 15 is 5 because 5 is the largest integer that evenly divides both 10 and 15.

There is a procedure or *algorithm* that can be followed to arrive at the *gcd* of two arbitrary integers. This algorithm is based on a procedure originally developed by Euclid around 300 B.C., and can be stated as follows:

Problem:	Find the greatest common divisor of two nonnegative integers u and v.
Step 1:	If v equals 0, then we are done and the *gcd* is equal to u.
Step 2:	Calculate $temp = u \% v$, $u = v$, $v = temp$ and go back to Step 1.

Don't concern yourself with the details of how the above algorithm works—simply take it on faith. We are more concerned here with developing the program to find the greatest common divisor than in performing an analysis of how the algorithm works.

Once the solution to the problem of finding the greatest common divisor has been expressed in terms of an algorithm, it becomes a much simpler task to develop the computer program. An analysis of the steps of the algorithm reveals that Step 2 is repetitively executed as long as the value of v is not equal to 0. This realization leads to the natural implementation of this algorithm in C with the use of a `while` statement.

The following program will find the *gcd* of two nonnegative integer values typed in by the user.

Program 5.7

```
/* This program finds the greatest common divisor
        of two nonnegative integer values        */
#include <stdio.h>
main ()
{
    int u, v, temp;

    printf ("Please type in two nonnegative integers.\n");
    scanf ("%i%i", &u, &v);
```

continues

Program 5.7 Continued

```
    while ( v != 0 )
    {
        temp = u % v;
        u = v;
        v = temp;
    }

    printf ("Their greatest common divisor is %i\n", u);
}
```

Program 5.7 OUTPUT

```
Please type in two nonnegative integers.
150 35
Their greatest common divisor is 5
```

Program 5.7 OUTPUT (Rerun)

```
Please type in two nonnegative integers.
1026 405
Their greatest common divisor is 27
```

The double `%i` characters in the `scanf` call indicate that two integer values are to be entered from the keyboard. The first value that is entered will be stored into the integer variable `u`, while the second will be stored into the variable `v`. When the values are actually entered from the terminal, they can be separated from each other by one or more blank spaces or by a carriage return. However, they cannot be separated by a comma (BASIC programmers beware).

After the values have been entered from the keyboard and stored into the variables `u` and `v`, the program enters a `while` loop to calculate their greatest common divisor. After the `while` loop is exited, the value of `u`, which represents the *gcd* of `v` and of the original value of `u`, is displayed at the terminal, together with an appropriate message.

For our next program to illustrate the use of the `while` statement, let us consider the task of reversing the digits of an integer that is entered from the terminal. For example, if the user types in the number 1234, we would like the program to reverse the digits of this number and display the result of 4321.

To write such a program, we first must come up with an algorithm that accomplishes the stated task. Frequently, an analysis of one's own method for solving the problem will lead to the development of an algorithm. For the task of reversing the digits of a number, the method of solution can be simply stated as "successively read the digits of the number from right to left." We can have a computer program "successively read" the digits of the number by developing a procedure to successively isolate or "extract" each digit of the number, beginning with the rightmost digit. The extracted digit can be subsequently displayed at the terminal as the next digit of the reversed number.

We can extract the rightmost digit from an integer number by taking the remainder of the integer after it is divided by 10. For example, 1234 % 10 will give the value 4, which is the rightmost digit of 1234, and is also the first digit of the reversed number. (Remember the modulus operator, which gives the remainder of one integer divided by another.) We can get the next digit of the number by using the same process if we first divide the number by 10, bearing in mind the way integer division works. Thus, 1234 / 10 gives a result of 123, and 123 % 10 gives us 3, which is the next digit of our reversed number.

This procedure can be continued until the last digit has been extracted. In the general case, we know that the last digit of the number has been extracted when the result of the last integer division by 10 is 0.

Program 5.8

```
/* Program to reverse the digits of a number */
#include <stdio.h>
main ()
{
    int  number, right_digit;

    printf ("Enter your number.\n");
    scanf ("%i", &number);

    while ( number != 0 )
    {
        right_digit = number % 10;
        printf ("%i", right_digit);
        number = number / 10;
    }

    printf ("\n");
}
```

Program 5.8 OUTPUT

```
Enter your number.
13579
97531
```

Each digit is displayed as it is extracted by the program. Notice that we did not include a newline character inside the `printf` statement contained in the `while` loop. This forces each successive digit to be displayed on the same line. The final `printf` call at the end of the program contains just a newline character, which causes the cursor to advance to the start of the next line.

The *do* Statement

The two looping constructs that we have discussed thus far in this chapter both make a test of the conditions *before* the loop is executed. Therefore, the body of the loop may never be executed at all if the conditions are not satisfied. When developing programs, it sometimes becomes desirable to have the test made at the *end* of the loop rather than at the beginning. Naturally, the C language provides a special language construct to handle such a situation. This looping statement is known as the `do` statement. The syntax of this statement is

```
do
     program statement
while ( expression );
```

Execution of the `do` statement proceeds as follows: *`program statement`* is executed first. Next, the *`expression`* inside the parentheses is evaluated. If the result of evaluating *`expression`* is TRUE, the loop continues and *`program statement`* is once again executed. As long as evaluation of *`expression`* continues to be TRUE, *`program statement`* is repeatedly executed. When evaluation of the expression proves FALSE, the loop is terminated, and the next statement in the program is executed in the normal sequential manner.

The `do` statement is simply a transposition of the `while` statement, with the looping conditions placed at the end of the loop rather than at the beginning.

QUICK TIP

Remember that, unlike the `for` and `while` loops, the `do` statement guarantees that the body of the loop will be executed at least once.

In Program 5.8, we used a `while` statement to reverse the digits of a number. Go back to that program and try to determine what would happen if the user had typed in the number 0 instead of 13579. The fact of the matter is that the loop of the `while` statement would never be executed and we would simply end up with a blank line in our display (as a result of the display of the newline character from the second `printf` statement). If we were to use a `do` statement instead of a `while`, then we would be assured that the program loop would be executed at least once, thus guaranteeing the display of at least one digit in all cases.

Program 5.9

```
/* Program to reverse the digits of a number */
#include <stdio.h>
main ()
{
     int  number, right_digit;

     printf ("Enter your number.\n");
     scanf ("%i", &number);

     do
     {
          right_digit = number % 10;
          printf ("%i", right_digit);
          number = number / 10;
     }
     while ( number != 0 );

     printf ("\n");
}
```

Program 5.9 OUTPUT

```
Enter your number.
13579
97531
```

Program 5.9 OUTPUT (Rerun)

```
Enter your number.
0
0
```

As you can see from the program's output, when 0 is keyed into the program, the program correctly displays the digit 0.

The *break* Statement

Sometimes when executing a loop it becomes desirable to leave the loop as soon as a certain condition occurs (for instance, maybe you detect an error condition, or you reach the end of your data prematurely). The `break` statement can be used for this purpose. Execution of the `break` statement causes the program to immediately exit from the loop it is executing, whether it's a `for`, `while`, or `do`. Subsequent statements in the loop are skipped, and execution of the loop is terminated. Execution continues with whatever statement follows the loop.

If a `break` is executed from within a set of nested loops, only the innermost loop in which the `break` is executed is terminated.

The format of the `break` statement is simply the keyword `break` followed by a semicolon:

```
break;
```

The *continue* Statement

The `continue` statement is similar to the `break` statement except it doesn't cause the loop to terminate. Rather, as its name implies, this statement causes the loop in which it is executed to be continued. At the point that the `continue` statement is executed, any statements in the loop that appear *after* the `continue` statement are automatically skipped. Execution of the loop otherwise continues as normal.

The `continue` statement is most often used to bypass a group of statements inside a loop based upon some condition, but to otherwise continue execution of the loop. The format of the `continue` statement is simply

```
continue;
```


QUICK TIP

Don't use the `break` or `continue` statements until you become very familiar with writing program loops and gracefully exiting from them. These statements are too easy to abuse and can result in programs that are hard to follow.

Now that you are familiar with all the basic looping constructs provided by the C language, we are ready to discuss another class of language statements that enable you to make decisions during the execution of a program. These decision-making capabilities are described in detail in the next chapter.

Exercises

1. If you have access to a computer facility that supports the C programming language, type in and run the nine programs presented in this chapter. Compare the output produced by each program with the output presented after each program.

2. Write a program to generate and display a table of n and n^2, for integer values of n ranging from 1 through 10. Be sure to print appropriate column headings.

3. A triangular number can also be generated by the formula

   ```
   Triangular number = n (n + 1) / 2
   ```

   for any integer value of n. For example, the tenth triangular number, 55, can be generated by substituting the 10 as the value for n into the above formula. Write a program that generates a table of triangular numbers using the above formula. Have the program generate every fifth triangular number between 5 and 50 (that is, 5, 10, 15, ..., 50).

4. The factorial of an integer n, written $n!$, is the product of the consecutive integers 1 through n. For example, 5 factorial is calculated as

   ```
   5!  =  5 x 4 x 3 x 2 x 1  =  120
   ```

   Write a program to generate and print a table of the first 10 factorials.

5. The following perfectly valid C program was written without much attention paid to its format. As you will observe, the program is not very readable. (And believe it or not, it is even possible to make this program significantly more unreadable!) Using the programs presented in this chapter as examples, re-format the program so that it is more readable. Then type the program into the computer and run it.

```
#include <stdio.h>
main(){
int n,two_to_the_n;
printf("TABLE OF POWERS OF TWO\n\n");
printf(" n     2 to the n\n");
printf("---    --------------\n");
two_to_the_n=1;
for(n=0;n<=10;++n){
printf("%2i        %i\n",n,two_to_the_n);
two_to_the_n=two_to_the_n*2;}
}
```

6. A minus sign placed in front of a field width specification causes the field to be displayed *left justified*. Substitute the following `printf` statement for the corresponding statement in Program 5.2, run the program, and compare the outputs produced by both programs.

```
printf ("%-2i            %i\n", n, triangular_number);
```

7. A decimal point before the field width specification in a `printf` statement has a special purpose. Try to determine its purpose by typing in and running the following program. Experiment by typing in different values each time you are prompted.

```
#include <stdio.h>
main ()
{
   int  dollars, cents, count;

   for ( count = 1;  count <= 10;  ++count )
   {
        printf ("Enter dollars: ");
        scanf ("%i", &dollars);
        printf ("Enter cents: ");
        scanf ("%i", &cents);
        printf ("$%i.%.2i\n\n", dollars, cents);
   }
}
```

8. Program 5.5 allows the user to type in only five different numbers. Modify that program so that the user can type in the number of triangular numbers to be calculated.

9. Rewrite Programs 5.2 through 5.5, replacing all uses of the `for` statement by equivalent `while` statements. Run each program to verify that

10. What would happen if you typed a negative number into Program 5.8? Try it and see.

11. Write a program that calculates the sum of the digits of an integer. For example, the sum of the digits of the number 2155 is 2 + 1 + 5 + 5 or 13. The program should accept any arbitrary integer typed in by the user.

6

CHAPTER

Making Decisions

In the previous chapter, we noted that one of the fundamental properties of a computer is its capability to repetitively execute a sequence of instructions. But another fundamental property lies in its capability to make decisions. We saw how these decision-making powers were used in the execution of the various looping statements to determine when to terminate the program loop. Without such capabilities, we would never be able to "get out" of a program loop and would end up executing the same sequence of statements over and over again, theoretically forever (which is why such a program loop is called an *infinite* loop).

The *if* Statement

The C programming language also provides a general decision-making capability in the form of a language construct known as the `if` statement. The general format of this statement is

```
if  ( expression )
      program statement
```

Imagine if you will that we could translate a statement such as "If it is not raining then I will go swimming" into the C language. Using the above format for the `if` statement, this might be "written" in C as follows:

```
if ( it is not raining )
    I will go swimming
```

The `if` statement is used to stipulate execution of a program statement (or statements if enclosed in braces) *based upon specified conditions*. I will go swimming if it is not raining. Similarly, in the program statement

```
if ( count > COUNT_LIMIT )
    printf ("Count limit exceeded\n");
```

the `printf` statement will be executed *only* if the value of `count` is greater than the value of `COUNT_LIMIT`; otherwise, it will be ignored.

An actual program example will help drive the point home. Suppose we wish to write a program that accepts an integer typed in from the terminal and then displays the absolute value of that integer. A straightforward way to calculate the absolute value of an integer is to simply negate the number if it is less than zero. The use of the phrase "if it is less than zero" in the previous sentence signals that a decision must be made by the program. This decision can be effected by the use of an `if` statement as shown in the program that follows.

Program 6.1

```
/* Calculate the absolute value of an integer */

main ()
{
    int  number;

    printf ("Type in your number: ");
    scanf ("%i", &number);

    if ( number < 0 )
        number = -number;

    printf ("The absolute value is %i\n", number);
}
```

Program 6.1 OUTPUT

```
Type in your number: -100
The absolute value is 100
```


Program 6.1 OUTPUT (Rerun)

```
Type in your number: 2000
The absolute value is 2000
```

The program was run twice to verify that it is functioning properly. Of course, it might be desirable to run the program several more times to get a higher level of confidence so that you know it is indeed working correctly, but at least we know that we have checked both possible outcomes of the decision made by the program.

After a message is displayed to the user and the integer value that is entered is stored into `number`, the program tests the value of `number` to see if it is less than zero. If it is, then the following program statement, which negates the value of `number`, is executed. If the value of `number` is not less than zero, then this program statement is automatically skipped. (If it is already positive, we don't want to negate it.) The absolute value of `number` is then displayed by the program, and program execution ends.

Let us now look at another program that uses the `if` statement. Imagine that we had a list of grades whose average we wished to compute. But in addition to computing the average, suppose that we also needed a count of the number of failing grades in the list. For the purposes of this problem, we can assume that a grade less than 65 is to be considered a failing grade.

The notion of keeping count of the number of failing grades indicates to us that we must make a decision as to whether a grade qualifies as a failing grade or not. Once again the `if` statement comes to the rescue.

Program 6.2

```
/* Program to calculate the average of a set of grades and to count
   the number of failing test grades     */
#include <stdio.h>
main ()
{
    int       number_of_grades, i, grade;
    int       grade_total = 0;
    int       failure_count = 0;
    float  average;

    printf ("How many grades will you be entering? ");
    scanf ("%i", &number_of_grades);
```

continues

Program 6.2 Continued

```
for ( i = 1;  i <= number_of_grades;  ++i )
{
    printf ("Enter grade #%i: ", i);
    scanf ("%i", &grade);

    grade_total = grade_total + grade;

    if ( grade < 65 )
        ++failure_count;
}

average = (float) grade_total / number_of_grades;

printf ("\nGrade average = %.2f\n", average);
printf ("Number of failures = %i\n", failure_count);
```

Program 6.2 OUTPUT

```
How many grades will you be entering? 7
Enter grade #1: 93
Enter grade #2: 63
Enter grade #3: 87
Enter grade #4: 65
Enter grade #5: 62
Enter grade #6: 88
Enter grade #7: 76

Grade average = 76.29
Number of failures = 2
```

The variable `grade_total`, which is used to keep a cumulative total of the grades as they are typed in at the terminal, is initially set to 0. The number of failing test grades will be stored in the variable `failure_count`, whose value also is initially set to 0. The variable `average` is declared to be of type `float`, since the average of a set of integers is not necessarily an integer itself.

The program then proceeds to ask the user to enter the number of grades that will be keyed in and stores the value that is entered into the variable `number_of_grades`. A loop is then set up that will be executed for each grade. The first part of the loop prompts the user to enter in the grade. The value that is entered is stored into the variable called, appropriately enough, `grade`.

The value of `grade` is then added into `grade_total`, after which a test is made to see if it is a failing test grade. If it is, then the value of `failure_count` is incremented by 1. The entire loop is then repeated for the next grade in the list.

When all of the grades have been entered and totaled, the program then proceeds to calculate the grade average. On impulse, it would seem that a statement such as

```
average = grade_total / number_of_grades;
```

would do the trick. However, you will recall that if the above statement were used, then the decimal portion of the result of the division would be lost. This would be due to the fact that an integer division would be performed, since *both* the numerator and the denominator of the division operation are integers.

There are really two different solutions to this problem. One would be to declare either `number_of_grades` or `grade_total` to be of type `float`. This would then guarantee that the division would be carried out without the loss of the decimal places. The only problem with this approach is that the variables `number_of_grades` and `grade_total` are used by the program to store only integer values. Declaring either of them to be of type `float` would only obscure their use in the program and is generally not a very clean way of doing things.

The other solution, as used by the program, is to actually *convert* the value of one of the variables to a floating-point value for the purposes of the calculation. The *type cast* operator `(float)` has the effect of converting the value of the variable `grade_total` to type `float` for purposes of evaluation of the expression. In no way does this operator permanently affect the value of the variable `grade_total`; it is a unary operator that behaves like other unary operators. And, as the expression `-a` has no permanent effect on the value of `a`, neither does the expression `(float) a`.

The type cast operator has a higher precedence than all the arithmetic operators except the unary minus and unary plus. Of course, if necessary, parentheses can always be used in an expression to force the terms to be evaluated in any desired order.

As another example of the use of the type cast operator, the expression

```
(int) 29.55 + (int) 21.99
```

will be evaluated in C as

```
29 + 21
```

since the effect of casting a floating value to an integer is one of truncating the floating-point value. The expression

```
(float) 6 / (float) 4
```

will produce a result of 1.5, as will the expression

```
(float) 6 / 4
```

Returning to our program, since the value of `grade_total` is cast into a floating-point value *before* the division takes place, the C system will treat the operation as the division of a floating value by an integer. Since one of the operands is now a floating-point value, the division operation will be carried out as a floating-point operation. This means, of course, that we will get those decimal places that we want in the average.

Once the average has been calculated, it is displayed at the terminal to two decimal places of accuracy. If a decimal point followed by a number (known collectively as a *precision modifier*) is placed directly before the format character `f` (or `e`) in a `printf` format string, the corresponding value will be displayed to the specified number of decimal places, rounded. So in Program 6.2, the precision modifier `.2` is used to cause the value of `average` to be displayed to two decimal places.

After the program has displayed the number of failing grades, execution of the program is complete.

The *if-else* Construct

If someone asks you whether a particular number is even or odd, you will most likely make the determination by examining the last digit of the number. If this digit is either 0, 2, 4, 6, or 8, then you will readily state that the number is even. Otherwise, you will claim that the number is odd.

An easier way for a computer to determine whether a particular number is even or odd is effected not by examining the last digit of the number to see if it is 0, 2, 4, 6, or 8, but by simply determining whether the number is evenly divisible by 2 or not. If it is, then the number is even; else it is odd.

We have already seen how the modulus operator `%` is used to compute the remainder of one integer divided by another. This makes it the perfect operator to use in determining whether or not an integer is evenly divisible by 2. If the remainder after division by 2 is 0, then it is even; else it is odd.

Let us now write a program that determines whether an integer value typed in by the user is even or odd and then displays an appropriate message at the terminal.

Program 6.3

```
/* Program to determine if a number is even or odd */
#include <stdio.h>
main ()
{
     int  number_to_test, remainder;

     printf ("Enter your number to be tested.: ");
     scanf ("%i", &number_to_test);

     remainder = number_to_test % 2;

     if ( remainder == 0 )
          printf ("The number is even.\n");

     if ( remainder != 0 )
          printf ("The number is odd.\n");
}
```

Program 6.3 OUTPUT

```
Enter your number to be tested: 2455
The number is odd.
```

Program 6.3 OUTPUT (Rerun)

```
Enter your number to be tested: 1210
The number is even.
```

After the number is typed in, the remainder after division by 2 is calculated. The first `if` statement tests the value of this remainder to see if it is equal to 0. If it is, the message "The number is even" is displayed.

The second `if` statement tests the remainder to see if it's *not* equal to 0 and, if that's the case, displays a message stating that the number is odd.

The fact is that whenever the first `if` statement succeeds, the second one must fail, and vice versa. If you recall from our discussions of even/odd numbers at the beginning of this section, we said that if the number is evenly divisible by 2, then it is even; *else* it is odd.

When writing programs, this "else" concept is so frequently required that almost all modern programming languages provide a special construct to handle this situation. In C, this is known as the `if-else` construct and the general format is as follows:

```
if  ( expression )
    program statement 1
else
    program statement 2
```

The `if-else` is actually just an extension of the general format of the `if` statement. If the result of the evaluation of the expression is TRUE, then *`program statement 1`*, which immediately follows, is executed; otherwise, *`program statement 2`* is executed. In either case, either *`program statement 1`* or *`program statement 2`* will be executed, but *not both*.

We can incorporate the `if-else` statement into the program above, replacing the two `if` statements by a single `if-else` statement. You will see how the use of this new program construct actually helps to reduce somewhat the program's complexity and also improves its readability.

Program 6.4

```
/* Determine if a number is even or odd (Ver. 2) */
#include <stdio.h>
main ()
{
        int  number_to_test, remainder;

        printf ("Enter your number to be tested: ");
        scanf ("%i", &number_to_test);

        remainder = number_to_test % 2;

        if ( remainder == 0 )
            printf ("The number is even.\n");
        else
            printf ("The number is odd.\n");
}
```

Program 6.4 OUTPUT

```
Enter your number to be tested: 1234
The number is even.
```


Program 6.4 OUTPUT (Rerun)

```
Enter your number to be tested: 6551
The number is odd.
```

QUICK TIP

Don't forget that the double equals sign == is the equality test and the single equals sign is the assignment operator. It can lead to lots of headaches if you forget this and inadvertently use the assignment operator inside the `if` statement.

Compound Relational Tests

The `if` statements that we have used so far in this chapter set up simple relational tests between two numbers. In Program 6.1, we compared the value of `number` against 0, while in Program 6.2, we compared the value of `grade` against 65. Sometimes it becomes desirable, if not necessary, to set up more sophisticated tests. Suppose, for example, that in Program 6.2 we wished to count not the number of failing grades, but instead the number of grades that were between 70 and 79, inclusive. In such a case, we would not merely wish to compare the value of `grade` against one limit, but against the two limits 70 and 79 to make sure that it fell within the specified range.

The C language provides the mechanisms necessary to perform these types of compound relational tests. A *compound relational test* is simply one or more simple relational tests joined by either the *logical AND* or the *logical OR* operator. These operators are represented by the character pairs `&&` and `||` (two vertical bar characters), respectively. As an example, the C statement

```
if  ( grade >= 70  &&  grade <= 79 )
    ++grades_70_to_79;
```

will increment the value of `grades_70_to_79` only if the value of `grade` is greater than or equal to 70 *and* less than or equal to 79. In a like manner, the statement

```
if  ( index < 0  ||  index > 99 )
    printf ("Error - index out of range\n");
```

will cause execution of the `printf` statement if `index` is less than 0 *or* greater than 99.

The compound operators can be used to form extremely complex expressions in C. The C language grants the programmer ultimate flexibility in forming expressions. This flexibility is a capability that is often abused. Simpler expressions are almost always easier to read and debug.

QUICK TIP

When forming compound relational expressions, liberally use parentheses to aid readability of the expression and to avoid getting into trouble because of a mistaken assumption about the precedence of the operators in the expression. (The `&&` operator has *lower* precedence than any arithmetic or relational operator but *higher* precedence than the `||` operator.) Blank spaces also should be used to aid in the expression's readability. An extra blank space around the `&&` and `||` operators will visually set these operators apart from the expressions that are being joined by these operators.

To illustrate the use of a compound relational test in an actual program example, let us write a program that tests to see whether a year is a leap year or not. We all know that a year is a leap year if it is evenly divisible by 4. What you may not realize, however, is that a year that is divisible by 100 is *not* a leap year unless it also is divisible by 400.

Let us try to think how we would go about setting up a test for such a condition. First, we could compute the remainders of the year after division by 4, 100, and 400, and assign these values to appropriately named variables, such as `rem_4`, `rem_100`, and `rem_400`, respectively. Then we could proceed to test these remainders to determine if the desired criteria for a leap year were met.

If we rephrase our definition of a leap year from above, we can say that a year is a leap year if it is evenly divisible by 4 and not by 100 or if it is evenly divisible by 400. Stop for a moment to reflect on this last sentence and to verify to yourself that it is equivalent to our previously stated definition. Now that we have reformulated our definition in these terms, it becomes a relatively straightforward task to translate it into a program statement as follows:

```
if ( (rem_4 == 0  &&  rem_100 != 0) ||  rem_400 == 0 )
    printf ("It's a leap year.\n");
```

The parentheses around the sub-expression

```
rem_4 == 0  &&  rem_100 != 0
```

are not required since that is how the expression will be evaluated anyway, remembering that `&&` has higher precedence than `||`.

(In fact, in this particular example, the test

```
if ( rem_4 == 0  &&  ( rem_100 != 0 ||  rem_400 == 0 ) )
```

would work just as well.)

If we add a few statements in front of our test to declare our variables and to enable the user to key in the year from the terminal, then we end up with a program that determines if a year is a leap year, as shown below.

Program 6.5

```
/* This program determines if a year is a leap year */
#include <stdio.h>
main ()
{
    int  year, rem_4, rem_100, rem_400;

    printf ("Enter the year to be tested: ");
    scanf ("%i", &year);

    rem_4 = year % 4;
    rem_100 = year % 100;
    rem_400 = year % 400;


    if ( (rem_4 == 0  &&  rem_100 != 0)  ||  rem_400 == 0 )
        printf ("It's a leap year.\n");
    else
        printf ("Nope, it's not a leap year.\n");
}
```

Program 6.5 OUTPUT

```
Enter the year to be tested: 1955
Nope, it's not a leap year.
```

Program 6.5 OUTPUT (Rerun)

```
Enter the year to be tested: 2000
It's a leap year.
```

Program 6.5 OUTPUT (Rerun)

```
Enter the year to be tested: 1800
Nope, it's not a leap year.
```

In the above examples, we used a year that was not a leap year because it wasn't evenly divisible by 4 (1955), a year that was a leap year because it was evenly divisible by 400 (2000), and a year that wasn't a leap year because it was evenly divisible by 100 but not by 400 (1800). To complete the run of test cases, we should also try a year that is evenly divisible by 4 but not by 100. This is left as an exercise for you.

We mentioned that C gives the programmer a tremendous amount of flexibility in forming expressions. For instance, in the above program, we did not have to calculate the intermediate results `rem_4`, `rem_100`, and `rem_400`—we could have performed the calculation directly inside the `if` statement as follows:

```
if ( ( year % 4 == 0  &&  year % 100 != 0 )  ||
               year % 400 == 0 )
```

The use of blank spaces to set off the various operators still makes the above expression readable. If we decided to ignore adding blanks and removed the unnecessary set of parentheses, we could end up with an expression that looked like this:

```
if(year%4==0&&year%100!=0)||year%400==0)
```

This expression is perfectly valid and would (believe it or not) execute identically to the expression shown immediately above it. Obviously, those extra blanks go a long way toward aiding our understanding of complex expressions.

Nested *if* Statements

In discussions of the general format of the `if` statement, we indicated that if the result of evaluating the expression inside the parentheses were TRUE, then the statement that immediately followed would be executed. It is perfectly valid that this program statement be another `if` statement, as in the statement

```
if ( game_is_over == 0 )
     if ( player_to_move == YOU )
          printf ("Your Move\n");
```

If the value of `game_is_over` is 0, then the following statement will be executed, which is in turn another `if` statement. This `if` statement will compare the value of `player_to_move` against `YOU`. If the two values are equal, then the message "Your Move" will be displayed at the terminal. Therefore, the `printf` statement will be executed only if `game_is_over` equals 0 *and* `player_to_move` equals `YOU`.

In fact, this statement could have been equivalently formulated using compound relationals as

```
if ( game_is_over == 0  &&  player_to_move == YOU )
     printf ("Your Move\n");
```

A more practical example of "nested" `if` statements would be if we added an `else` clause to the above example, as shown in the following:

```
if ( game_is_over == 0 )
     if ( player_to_move == YOU )
          printf ("Your Move\n");
     else
          printf ("My Move\n");
```

Execution of this statement proceeds as described above. However, if `game_is_over` equals 0 and the value of `player_to_move` is not equal to `YOU`, then the `else` clause will be executed. This will display the message "My Move" at the terminal. If `game_is_over` does not equal 0, the entire `if` statement that follows, including its associated `else` clause, will be skipped.

Notice how the `else` clause is associated with the `if` statement that tests the value of `player_to_move`, and not with the `if` statement that tests the value of `game_is_over`. The general rule is that an `else` clause is always associated with the last `if` statement that does not contain an `else`.

We can go one step further and can add an `else` clause to the outermost `if` statement in the preceding example. This `else` clause would be executed if the value of `game_is_over` is not 0.

```
if ( game_is_over == 0 )
     if ( player_to_move == YOU )
          printf ("Your Move\n");
     else
          printf ("My Move\n");
else
     printf ("The game is over\n");
```

QUICK TIP

The proper use of indentation goes a long way toward aiding your understanding of the logic of complex statements.

Of course, even if we use indentation to indicate the way we think a statement will be interpreted in the C language, it may not always coincide with the way that the system will actually interpret the statement. For instance, removing the first `else` clause from the above example

```
if ( game_is_over == 0 )
     if ( player_to_move == YOU )
          printf ("Your Move\n");
else
     printf ("The game is over\n");
```

will *not* result in the statement's being interpreted as indicated by its format. Instead, this statement will be interpreted as

```
if ( game_is_over == 0 )
     if ( player_to_move == YOU )
          printf ("Your Move\n");
     else
          printf ("The game is over\n");
```

since the `else` clause is associated with the last un-elsed `if`. We could use braces to force a different association in those cases where an innermost `if` does not contain an `else`, but an outer `if` does. The braces have the effect of "closing off" the `if` statement. Thus,

```
if ( game_is_over == 0 )
{
     if ( player_to_move == YOU )
          printf ("Your Move\n");
}
else
     printf ("The game is over\n");
```

will achieve the desired effect, with the message "The game is over" being displayed if the value of `game_is_over` is not 0.

The *else if* Construct

We have seen how the `else` statement comes into play when we have a test against two possible conditions—either the number is even, else it is odd; either the year is a leap year, else it is not. However, programming decisions that we have to make are not always so black-and-white. Consider the task of writing a program that displayed -1 if a number typed in by a user were less than zero, 0 if the number typed in were equal to zero, and 1 if the number were greater than zero. (This is actually an implementation of what is commonly called the *sign* function.) Obviously, we must make three tests in this case—to determine if the number that is keyed in is negative, zero, or positive. Our simple `if-else` construct will not work. Of course, in this case, we could always resort to three separate `if` statements, but this solution will not always work in general—especially if the tests that are made are not mutually exclusive.

We can handle the situation just described by adding an `if` statement to our `else` clause. We mentioned that the statement that followed an `else` could be any valid C program statement, so why not another `if`? Thus, in the general case, we could write


```
if ( expression 1 )
     program statement 1
else
     if ( expression 2 )
          program statement 2
     else
program statement 3
```

which effectively extends the `if` statement from a two-valued logic decision to a three-valued logic decision. We can continue to add `if` statements to the `else` clauses, in the manner just shown, to effectively extend the decision to an *n*-valued logic decision.

The preceding construct is so frequently used that it is generally referred to as an `else if` construct and is usually formatted differently from that shown previously as

```
if ( expression 1 )
     program statement 1
else if ( expression 2 )
     program statement 2
else
     program statement 3
```

This latter method of formatting improves the readability of the statement and makes it clearer that a three-way decision is being made.

The next program illustrates the use of the `else if` construct by implementing the sign function discussed earlier.

Program 6.6

```
/* Program to implement the sign function */
#include <stdio.h>
main ()
{
     int  number, sign;

     printf ("Please type in a number: ");
     scanf ("%i", &number);

     if ( number < 0 )
          sign = -1;
     else if ( number == 0 )
          sign = 0;
     else          /* Must be positive */
          sign = 1;

     printf ("Sign = %i\n", sign);
}
```

Program 6.6 OUTPUT

```
Please type in a number: 1121
Sign = 1
```

Program 6.6 OUTPUT (Rerun)

```
Please type in a number: -158
Sign = -1
```

Program 6.6 OUTPUT (Rerun)

```
Please type in a number: 60
Sign = 0
```

If the number that is entered is less than zero, `sign` is assigned the value -1; if the number is equal to zero, `sign` is assigned the value 0; otherwise, the number must be greater than zero, so `sign` is assigned the value 1.

The next program analyzes a character that is typed in from the terminal and classifies it as either an alphabetic character (*a-z* or *A-Z*), a digit (*0-9*), or a special character (anything else). In order to read a single character from the terminal, the format characters `%c` are used in the `scanf` call.

Program 6.7

```
/* This program categorizes a single character
            that is entered at the terminal          */
#include <stdio.h>
main ()
{
    char  c;

    printf ("Enter a single character:\n");
    scanf ("%c", &c);

    if ( (c >= 'a'  &&  c <= 'z') || (c >= 'A'  &&  c <= 'Z') )
        printf ("It's an alphabetic character.\n");
    else if  ( c >= '0'  &&  c <= '9' )
```



```
        printf ("It's a digit.\n");
    else
        printf ("It's a special character.\n");
}
```

Program 6.7 OUTPUT

```
Enter a single character:
&
It's a special character.
```

Program 6.7 OUTPUT (Rerun)

```
Enter a single character:
8
It's a digit.
```

Program 6.7 OUTPUT (Rerun)

```
Enter a single character:
B
It's an alphabetic character.
```

The first test that is made after the character is read in determines if the `char` variable `c` is an alphabetic character or not. This is done by testing if the character is either a lowercase letter or an uppercase letter. The former test is made by the expression

```
( c >= 'a'  &&  c <= 'z' )
```

which will be TRUE if `c` is within the range of characters `'a'` through `'z'`; that is, if `c` is a lowercase letter. The latter test is made by the expression

```
( c >= 'A'  &&  c <= 'Z' )
```

which will be TRUE if `c` is within the range of characters `'A'` through `'Z'`; that is, if `c` is an uppercase letter. These tests work on all computer systems that store characters inside the machine in a format known as ASCII format. However, they do not work correctly on machines which use the EBCDIC format, since there are characters other than letters which fall within the tested ranges.

If the variable c is an alphabetic character, then the first `if` test will succeed and the message "It's an alphabetic character" will be displayed. If the test fails, then the `else if` clause will be executed. This clause determines if the character is a digit. Note that this test compares the character c against the *characters* `'0'` and `'9'` and *not* the *integers* 0 and 9. This is because a character was read in from the terminal, and the characters `'0'` to `'9'` are not the same as the numbers 0–9. In fact, on a computer system that uses the ASCII format mentioned above, the character `'0'` is actually represented internally as the number 48, the character `'1'` as the number 49, and so on.

If c is a digit character, then the phrase "It's a digit" will be displayed. Otherwise, if c is not alphabetic and is not a digit, then the final `else` clause will be executed and will display the phrase "It's a special character." Execution of the program will then be complete.

You should note that even though `scanf` is used here to read just a single character, the Return key must still be pressed after the character is typed to send the input to the program. In general, whenever you're reading data from the terminal, the program doesn't see any of the data typed on the line until the Return key is pressed.

Let us suppose for our next example that we wished to write a program that allowed the user to type in simple expressions of the form

number operator number

The program will evaluate the expression and display the results at the terminal, to two decimal places of accuracy. The operators that we want to have recognized are the normal operators for addition, subtraction, multiplication, and division. The following program makes use of a large `if` statement with many `else if` clauses to determine which operation is to be performed.

Program 6.8

```
/* Program to evaluate simple expressions of the form
                number  operator  number                    */
#include <stdio.h>
main ()
{
    float  value1, value2;
    char   operator;

    printf ("Type in your expression.\n");
    scanf ("%f %c %f", &value1, &operator, &value2);

    if ( operator == '+' )
        printf ("%.2f\n", value1 + value2);
    else if ( operator == '-' )
        printf ("%.2f\n", value1 - value2);
    else if ( operator == '*' )
        printf ("%.2f\n", value1 * value2);
    else if ( operator == '/' )
        printf ("%.2f\n", value1 / value2);
}
```


Program 6.8 OUTPUT

```
Type in your expression.
123.5 + 59.3
182.80
```

Program 6.8 OUTPUT (Rerun)

```
Type in your expression.
198.7 / 26
7.64
```

Program 6.8 OUTPUT (Rerun)

```
Type in your expression.
89.3 * 2.5
223.25
```

The `scanf` call specifies that three values are to be read into the variables `value1`, `operator`, and `value2`. A floating value can be read in with the `%f` format characters, the same characters used for the output of floating values. This is the format used to read in the value of the variable `value1`, which is the first operand of our expression.

Next, we wish to read in the operator. Since the operator is a character (`'+'`, `'-'`, `'*'`, or `'/'`) and not a number, we read it into the character variable `operator`. The `%c` format characters tell the system to read in the next character from the terminal. The blank spaces inside the format string indicate that an arbitrary number of blank spaces are to be permitted on the input. This enables us to separate the operands from the operator with blank spaces when we type in these values. If we had specified the format string `"%f%c%f"` instead, then no

spaces would have been permitted after typing in the first number and before typing in the operator. This is because when the `scanf` function is reading a character with the `%c` format characters, the next character on the input, *even if it is a blank space*, is the character that is read. However, it should be noted that, in general, the `scanf` function will *always* ignore leading spaces when it is reading in either a decimal or floating-point number. Therefore, the format string `"%f %c%f"` would have worked just as well in the preceding program.

After the second operand has been keyed in and stored in the variable `value2`, the program proceeds to test the value of `operator` against the four permissible operators. When a correct match is made, the corresponding `printf` statement is executed to display the results of the calculation. Execution of the program is then complete.

A few words about program thoroughness are in order at this point. While the preceding program does accomplish the task that we set out to perform, the program is not really complete since it does not account for mistakes made on the part of the user. For example, what would happen if the user were to type in a ? for the operator by mistake? The program would simply "fall through" the `if` statement and no messages would ever appear at the terminal to alert the user that he had incorrectly typed in his expression.

Another case that is overlooked is when the user types in a division operation with 0 as the divisor. We know by now that we should never attempt to divide a number by 0 in C. The program should check for this case.

Trying to predict the ways that a program can fail or produce unwanted results and then taking preventive measures to account for such situations is a necessary part of producing good, reliable programs. Running a sufficient number of test cases against a program will often point the finger to portions of the program that do not account for certain cases. But it goes further than that. It must become a matter of self-discipline while coding a program to always say "What would happen if ..." and to insert the necessary program statements to handle the situation properly.

Program 6.8A, a modified version of Program 6.8, accounts for division by 0 and the keying in of an unknown operator.

Program 6.8A

```
/* Program to evaluate simple expressions of the form
                value   operator   value                    */
#include <stdio.h>
main ()
{
    float  value1, value2;
    char   operator;

    printf ("Type in your expression.\n");
    scanf ("%f %c %f", &value1, &operator, &value2);

    if ( operator == '+' )
         printf ("%.2f\n", value1 + value2);
    else if ( operator == '-' )
         printf ("%.2f\n", value1 - value2);
    else if ( operator == '*' )
         printf ("%.2f\n", value1 * value2);
    else if ( operator == '/' )
         if ( value2 == 0 )
              printf ("Division by zero.\n");
         else
              printf ("%.2f\n", value1 / value2);
    else
         printf ("Unknown operator.\n");
}
```


Program 6.8A OUTPUT

```
Type in your expression.
123.5 + 59.3
182.80
```

Program 6.8A OUTPUT (Rerun)

```
Type in your expression.
198.7 / 0
Division by zero.
```

Program 6.8A OUTPUT (Rerun)

```
Type in your expression.
125 $ 28
Unknown operator.
```

When the operator that is typed in is the slash, for division, another test is made to determine if *value2* is 0. If it is, an appropriate message is displayed at the terminal. Otherwise, the division operation is carried out and the results displayed. Pay careful attention to the nesting of the `if` statements and the associated `else` clauses in this case.

The `else` clause at the end of the program catches any "fall throughs." Therefore, any value of *operator* that does not match any of the four characters tested will cause this `else` clause to be executed, resulting in the display of "Unknown operator."

The *switch* Statement

The type of `if-else` statement chain that we encountered in the last program example—where the value of a variable is successively compared against different values—is so commonly used when developing programs that a special program statement exists in the C language for performing precisely this function. The name of the statement is the `switch` statement, and its general format is:

```
switch ( expression )
{
    case value1:
            program statement
            program statement
                ...
            break;
    case value2:
            program statement
            program statement
                ...
            break;
        ...
    case valuen:
            program statement
            program statement
                ...
            break;
    default:
            program statement
            program statement
                ...
            break;
}
```

The *expression* enclosed within parentheses is successively compared against the values *value1*, *value2*, ..., *valuen*, which must be simple constants or constant expressions. If a case is found whose value is equal to the value of *expression*, then the program statements that follow the case are executed. You

will note that when more than one such program statement is included, they do not have to be enclosed within braces.

The `break` statement signals the end of a particular case and causes execution of the `switch` statement to be terminated.

> **QUICK TIP**
>
> Remember to include the `break` statement at the end of every case. Forgetting to do so for a particular case will cause program execution to continue into the next case whenever that case gets executed.

The special optional case called `default` is executed if the value of `expression` does not match any of the case values. This is conceptually equivalent to the "fall through" `else` that we used in the previous example. In fact, the general form of the `switch` statement can be equivalently expressed as an `if` statement as follows:

```
if ( expression == value1 )
{
        program statement
        program statement
            ...
}
else if ( expression == value2 )
{
        program statement
        program statement
            ...
}
    ...
else if ( expression == valuen )
{
        program statement
        program statement
            ...
}
else
{
        program statement
        program statement
            ...
}
```

Bearing the above in mind, we can translate the big `if` statement from Program 6.8A into an equivalent `switch` statement. We will call this new program Program 6.9.

Program 6.9

```
/* Program to evaluate simple expressions of the form
            value  operator   value                    */
#include <stdio.h>
main ()
{
   float  value1, value2;
   char   operator;

   printf ("Type in your expression.\n");
   scanf ("%f %c %f", &value1, &operator, &value2);

   switch ( operator )
   {
      case '+':
         printf ("%.2f\n", value1 + value2);
         break;
      case '-':
         printf ("%.2f\n", value1 - value2);
         break;
      case '*':
         printf ("%.2f\n", value1 * value2);
         break;
      case '/':
         if ( value2 == 0 )
            printf ("Division by zero.\n");
         else
            printf ("%.2f\n", value1 / value2); break;
      default:
         printf ("Unknown operator.\n");
         break;
   }
}
```

Program 6.9 OUTPUT

```
Type in your expression.
178.99 - 326.8
-147.81
```

After the expression has been read in, the value of `operator` is successively compared against the values as specified by each case. When a match is found, the statements contained inside the case will be executed. The `break` statement

will then send execution out of the `switch` statement, where execution of the program will be complete. If none of the cases match the value of `operator`, the `default` case, which displays "Unknown operator", will be executed.

The `break` statement in the `default` case is actually unnecessary in the preceding program since no statements follow this case inside the `switch`. Nevertheless, it is a good programming habit to remember to include the `break` at the end of every case.

When writing a `switch` statement, you should bear in mind that no two case values may be the same. However, we can associate more than one case value with a particular set of program statements. This is done simply by listing the multiple case values (with the keyword `case` before the value and the colon after the value in each case) before the common statements that are to be executed. As an example, in the `switch` statement

```
switch  ( operator )
{
                ...
     case '*':
     case 'x':
               printf ("%.2f\n", value1 * value2);
               break;
                ...
}
```

the `printf` statement, which multiplies `value1` by `value2`, will be executed if `operator` is equal to an asterisk or to the lowercase letter `x`.

Flags

Just about all new programmers soon find themselves with the task of having to write a program to generate a table of *prime numbers*. To refresh your memory, a positive integer p is a prime number if it not evenly divisible by any other integers, other than 1 and itself. The first prime integer is defined to be 2. The next prime is 3, since it is not evenly divisible by any integers other than 1 and 3; and 4 is *not* prime because it *is* evenly divisible by 2.

There are several approaches that we could take in order to generate a table of prime numbers. If we had the task of generating all prime numbers up to 50, for example, then the most straightforward (and simplest) algorithm to generate such a table would simply test each integer p for divisibility by all integers from 2 through p-1. If any such integer evenly divided p, then p would not be prime; otherwise, it would be a prime number.

Program 6.10

```
/* Program to generate a table of prime numbers */
#include <stdio.h>

main ()
{
        int  p, is_prime, d;

        for ( p = 2;  p <= 50;  ++p )
        {
            is_prime = 1;

            for ( d = 2;  d < p;  ++d )
                if ( p % d  ==  0 )
                    is_prime = 0;

            if ( is_prime != 0 )
                printf ("%i  ", p);
        }


        printf ("\n");
}
```

Program 6.10 OUTPUT

```
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47
```

Several points are worth noting about the above program. The outermost `for` statement sets up a loop to cycle through the integers 2 through 50. The loop variable `p` represents the value we are currently testing to see if it is prime. The first statement in the loop assigns the value 1 to the variable `is_prime`. The use of this variable will become apparent shortly.

A second loop is set up to divide `p` by the integers from 2 through `p`-1. Inside the loop, a test is made to see if the remainder of `p` divided by `d` is 0. If it is, then we know that `p` cannot be prime since there exists an integer other than 1 and itself that evenly divides it. To signal that `p` is no longer a candidate as a prime number, the value of the variable `is_prime` is set equal to 0.

When the innermost loop finishes execution, the value of `is_prime` is tested. If its value is not equal to zero, no integer was found that evenly divided `p`; therefore, `p` must be a prime number, and its value is displayed.

You may have noticed that the variable `is_prime` takes on either the value 0 or 1, and no other values. Its value is 1 as long as `p` still qualifies as a prime number. But as soon as a single even divisor is found, its value is set to 0 to indicate that `p` no longer satisfies the criteria for being prime. Variables that are used in such a manner are generally referred to as Boolean variables, or more simply, as *flags*. A flag will typically assume only one of two different values. Furthermore, the value of a flag will usually be tested at least once in the program to see if it is "on" (TRUE) or "off" (FALSE) and some particular action taken based upon the results of the test.

In C, the notion of a flag being TRUE or FALSE is most naturally translated into the values 1 and 0, respectively. So in the above program, when we set the value of `is_prime` to 1 inside the loop, we are effectively setting it TRUE to indicate that `p` "is prime." If during the course of execution of the inner `for` loop an even divisor is found, the value of `is_prime` is set FALSE to indicate that `p` no longer "is prime."

It is no coincidence that the value 1 is typically used to represent the TRUE or "on" state and 0 to represent the FALSE or "off" state. This representation corresponds to the notion of a single bit inside a computer. When the bit is "on," its value is 1; when it is "off," its value is 0. But in C, there is an even more convincing argument in favor of these logic values. It has to do with the way the C language treats the concept of TRUE and FALSE.

When we began our discussions in this chapter, we noted that if the conditions specified inside the `if` statement were "satisfied," then the program statement that immediately followed would be executed. But what exactly does "satisfied" mean? In the C language, satisfied means nonzero, and nothing more. So the statement

```
if ( 100 )
     printf ("This will always be printed.\n");
```

will result in execution of the `printf` statement because the condition in the `if` statement (in this case simply the value 100) is nonzero and therefore is satisfied.

In each of the programs in this chapter, the notions of "nonzero means satisfied" and "zero means not satisfied" were used. This is because whenever a relational expression is evaluated in C, it is given the value 1 if the expression is satisfied and 0 if the expression is not satisfied. So evaluation of the statement

```
if  ( number < 0 )
     number = -number;
```

actually proceeds as follows:

1. The relational expression `number < 0` is evaluated. If the condition is satisfied, that is, if `number` is less than 0, then the value of the expression is 1; otherwise, its value is 0.

2. The `if` statement tests the result of the expression evaluation. If the result is nonzero, then the statement that immediately follows is executed; otherwise, the statement is skipped.

The preceding discussion also applies to evaluation of conditions inside the `for`, `while`, and `do` statements. Evaluation of compound relational expressions such as in the statement

```
while  ( char != 'e'  &&  count != 80 )
```

also proceeds as outlined above. If both specified conditions are valid, then the result will be 1; but if either condition is not valid, then the result of the evaluation will be 0. The results of the evaluation will then be checked. If the result is 0, then the `while` loop will terminate; otherwise it will continue.

Returning to Program 6.10 and the notion of flags, it is perfectly valid in C—and even clearer—to test if the value of a flag is TRUE by an expression such as

```
if ( is_prime )
```

rather than with the equivalent expression

```
if ( is_prime != 0 )
```

To easily test if the value of a flag is FALSE, we bring into play the *logical negation* operator, `!`. In the expression

```
if ( ! is_prime )
```

the logical negation operator is used to test if the value of `is_prime` is FALSE (read this statement as "if not `is_prime`"). In general, an expression such as

```
! expression
```

negates the logical value of *`expression`*. So if *`expression`* is 0, the logical negation operator produces a 1. And if the result of the evaluation of *`expression`* is nonzero, the negation operator yields a 0.

The logical negation operator can be used to easily "flip" the value of a flag, such as in the expression

```
my_move = ! my_move;
```

As you might expect, this operator has the same precedence as the unary minus operator, which means that it has higher precedence than all binary

arithmetic operators and all relational operators. So to test if the value of a variable `x` is not less than the value of a variable `y`, such as in

```
! ( x < y )
```

the parentheses are required to ensure proper evaluation of the expression. Of course, we could have equivalently expressed the above as

```
x >= y
```

The Conditional Operator

Perhaps the most unusual operator in the C language is one called the *conditional* operator. Unlike all other operators in C—which are either unary or binary operators—the conditional operator is a *ternary* operator; that is, it takes three operands. The two symbols that are used to denote this operator are the question mark (`?`) and the colon (`:`). The first operand is placed before the `?`, the second between the `?` and the `:`, and the third after the `:`.

The general format of the conditional operator is

```
condition  ?  expression1  :  expression2
```

where *`condition`* is an expression, usually a relational expression, that is evaluated by the C system first whenever the conditional operator is encountered. If the result of the evaluation of *`condition`* is TRUE (that is, nonzero), then *`expression1`* is evaluated and the result of the evaluation becomes the result of the operation. If *`condition`* evaluates FALSE (that is, zero), then *`expression2`* is evaluated and its result becomes the result of the operation.

The conditional operator is most often used to assign one of two values to a variable depending upon some condition. For example, suppose we have an integer variable `x` and another integer variable `s`. If we wished to assign -1 to `s` if `x` were less than 0, and the value of x^2 to `s` otherwise, the following statement could be written:

```
s = ( x < 0 ) ? -1 : x * x;
```

The condition `x < 0` is first tested when the above statement is executed. Parentheses are generally placed around the condition expression to aid in the statement's readability. This is usually not required, since the precedence of the conditional operator is very low—lower, in fact, than all other operators but the assignment operators and the comma operator.

If the value of `x` is less than zero, then the expression immediately following the `?` will be evaluated. This expression is simply the constant integer value -1, which will be assigned to the variable `s` if `x` is less than zero.

If the value of x is not less than zero, then the expression immediately following the : will be evaluated and assigned to s. So if x is greater than or equal to zero, the value of `x * x`, or x^2, will be assigned to s.

As another example of the use of the conditional operator, the following statement assigns to the variable `max_value` the maximum of a and b.

```
max_value = ( a > b ) ? a : b;
```

If the expression that is used after the : (the "else" part) consists of another conditional operator, then we can achieve the effects of an "else if" clause. For example, the *sign* function that was implemented in Program 6.6 can be written in one program line using two conditional operators as follows:

```
sign = ( number < 0 ) ? -1 : (( number == 0 ) ? 0 : 1);
```

If `number` is less than zero, then `sign` is assigned the value -1; else if `number` is equal to zero, then `sign` is assigned the value 0; else it is assigned the value 1. The parentheses around the "else" part of the above expression are actually unnecessary. This is because the conditional operator associates from right to left, meaning that multiple uses of this operator in a single expression, such as in

```
e1 ? e2 : e3 ? e4 : e5
```

will group from right to left and, therefore, will be evaluated as

```
e1 ? e2 : ( e3 ? e4 : e5 )
```

It is not necessary that the conditional operator be used on the right-hand side of an assignment—it can be used in any situation where an expression could be used. This means that we could display the sign of the variable `number`, without first assigning it to a variable, using a `printf` statement as shown:

```
printf ("Sign = %i\n", ( number < 0 ) ? -1
    : ( number == 0 ) ? 0 : 1);
```

The conditional operator is very handy when writing preprocessor *macros* in C. This will be seen in detail in Chapter 13, "The Preprocessor."

This concludes our discussions on making decisions. In the next chapter you will get your first look at more sophisticated data types. The *array* is a powerful concept that will find its way into many programs that you will develop in C.

Exercises

1. If you have access to a computer facility that supports the C programming language, type in and run the 10 programs presented in this chapter. Compare the output produced by each program with the output presented after each program. Try experimenting with each program by keying in values other than those shown.

2. Write a program that asks the user to type in two integer values at the terminal. Test these two numbers to determine if the first is evenly divisible by the second, and then display an appropriate message at the terminal.

3. Write a program that accepts two integer values typed in by the user. Display the result of dividing the first integer by the second, to three-decimal-place accuracy. Remember to have the program check for division by 0.

4. Write a program that acts as a simple "printing" calculator. The program should allow the user to type in expressions of the form

   ```
   number   operator
   ```

   The following operators should be recognized by the program:

   ```
   +    -    *    /    S    E
   ```

   The `S` operator tells the program to set the "accumulator" to the typed-in number. The `E` operator tells the program that execution is to end. The arithmetic operations are performed on the contents of the accumulator with the number that was keyed in acting as the second operand. The following is a "sample run" showing how the program should operate:

```
Begin Calculations
10 S                          Set Accumulator to 10
= 10.000000                   Contents of Accumulator
2 /                           Divide by 2
= 5.000000                    Contents of Accumulator
55 -                          Subtract 55
-50.000000
100.25 S                      Set Accumulator to 100.25
= 100.250000
4 *                           Multiply by 4
= 401.000000
0 E                           End of program
= 401.000000
End of Calculations.
```

   Make sure that the program detects division by 0 and also checks for unknown operators.

5. We developed Program 5.9 to reverse the digits of an integer typed in from the terminal. However, this program does not function too well if we type in a negative number. Find out what happens in such a case and then modify the program so that negative numbers are correctly handled. By correctly handled, we mean that if the number -8645 were typed in, for example, then the output of the program should be 5468-.

6. Write a program that takes an integer keyed in from the terminal and extracts and displays each digit of the integer in English. So, if the user types in 932, then the program should display

```
nine three two
```

(Remember to display "zero" if the user types in just a 0.) Note: This exercise is a hard one!

7. Program 6.10 has several inefficiencies. One inefficiency results from checking even numbers. Since it is obvious that any even number greater than 2 cannot be prime, the program could simply skip all even numbers as possible primes *and* as possible divisors. The inner `for` loop is also inefficient because the value of `p` is *always* divided by all values of `d` from 2 through `p`-1. This inefficiency could be avoided if we added a test for the value of `is_prime` in the conditions of the `for` loop. In this manner, the `for` loop could be set up to continue as long as no divisor was found and the value of `d` was less than `p`. Modify Program 6.10 to incorporate these two changes. Then run the program to verify its operation. Note: In the next chapter, we will find even more efficient ways of generating prime numbers.

7

CHAPTER

Arrays

The C language provides a capability that enables the user to define a set of ordered data items known as an *array*. This chapter describes how arrays can be defined and manipulated in C. In later chapters, we will include further discussions on arrays to illustrate how well they work together with program functions, structures, character strings, and pointers.

Suppose we had a set of `grades` that we wished to read into the computer, and suppose that we wished to perform some operations on these `grades`, such as rank them in ascending order, compute their average, or find their median. In Program 6.2, we were able to calculate the average of a set of `grades` by simply adding each grade into a cumulative total as each grade was keyed in. However, if we wanted to rank the `grades` into ascending order, for example, then we would have to do something further. If you think about the process of ranking a set of `grades`, you will quickly realize that we cannot perform such an operation until each and every grade has been entered. Therefore, using the techniques we have already described, we would read in each grade and store it into a unique variable, perhaps with a sequence of statements such as

```
printf ("Enter grade 1\n");
scanf ("%i", &grade1);
```

```
grades[100] = 95;
```

the value 95 is stored into element number 100 of the `grades` array. The statement

```
grades[i] = g;
```

will have the effect of storing the value of `g` into `grades[i]`.

The capability to represent a collection of related data items by a single array enables us to develop concise and efficient programs. For example, we can easily sequence through the elements in the array by varying the value of a variable that is used as a subscript into the array. So the `for` loop

```
for ( i = 0;  i < 100;  ++i )
      sum = sum + grades[i];
```

will sequence through the first 100 elements of the array `grades` (elements 0 through 99) and will add the value of each grade into `sum`. When the `for` loop is finished, the variable `sum` will then contain the total of the first 100 values of the `grades` array (assuming `sum` were set to 0 before the loop was entered).

> **QUICK TIP**
>
> Don't forget that the first element of an array is indexed by zero, and the last element by the number of elements in the array minus one.

In addition to integer constants, integer-valued expressions can also be used inside the brackets to reference a particular element of an array. So if `low` and `high` were defined as integer variables, then the statement

```
next_value = sorted_data[(low + high) / 2];
```

would assign to the variable `next_value` the value indexed by evaluating the expression `(low + high) / 2`. If `low` were equal to 1 and `high` were equal to 9, then the value of `sorted_data[5]` would be assigned to `next_value`. And if `low` were equal to 1 and `high` were equal to 10, then the value of `sorted_data[5]` would also be referenced, since we know that an integer division of 11 by 2 gives the result of 5.

Just as with variables, arrays must also be declared before they are used. The declaration of an array involves declaring the type of element that will be contained in the array—such as `int`, `float`, or `char`—as well as the maximum number of elements that will be stored inside the array. (The C system needs this latter information in order to determine how much of its memory space to reserve for the particular array.)

As an example, the declaration

```
int  grades[100];
```

declares grades to be an array containing 100 integer elements. Valid references to this array may be made by using subscripts from 0 through 99. But be careful to use valid subscripts, since C does not do any checking of array bounds for you. So a reference to element number 150 of array grades, as previously declared, would not necessarily cause an error but would most likely cause unwanted, if not unpredictable, program results.

To declare an array called averages that contained 200 floating-point elements, the declaration

```
float  averages[200];
```

would be used. This declaration would cause enough space inside the computer's memory to be reserved to contain 200 floating-point numbers. Similarly, the declaration

```
int  values[10];
```

would reserve enough space for an array called values that could hold up to 10 integer numbers. We could better conceptualize this reserved storage space by referring to Figure 7.1.

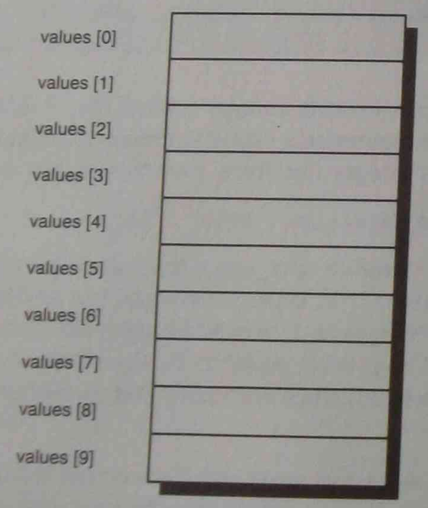


Figure 7.1. The array values in memory.

The elements of arrays declared to be of type int, float, or char may be manipulated in the same fashion as ordinary variables. We can assign values to

them, display their values, add to them, subtract from them, and so on. So if the following statements were to appear in a program

```
int  values[10];

values[0] = 197;
values[2] = -100;
values[5] = 350;
values[3] = values[0] + values[5];
values[9] = values[5] / 10;
--values[2];
```

then the array values would contain the numbers as shown in Figure 7.2.

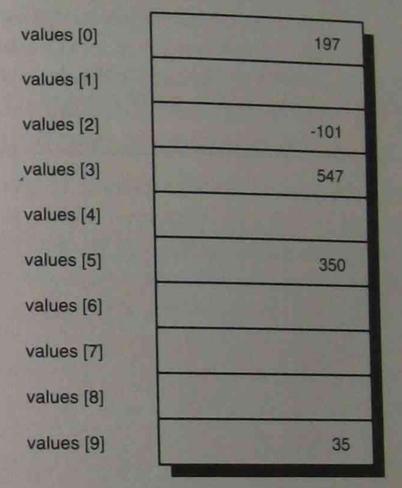


Figure 7.2. values *with some initialized elements.*

The first assignment statement has the effect of storing the value of 197 into values[0]. In a similar fashion, the second and third assignment statements store -100 and 350 into values[2] and values[5], respectively. The next statement adds the contents of values[0] (which is 197) to the contents of values[5] (which is 350) and stores the result of 547 in values[3]. In the following program statement, 350—the value contained in values[5]—is divided by 10 and the result stored into values[9]. The last statement decrements the contents of values[2], which has the effect of changing its value from -100 to -101.

The preceding program statements were incorporated into Program 7.1. The for loop sequences through each element of the array, displaying its value at the terminal in turn.

Program 7.1

```
#include <stdio.h>
main ()
{
        int  values[10];
        int  index;

        values[0] = 197;
        values[2] = -100;
        values[5] = 350;
        values[3] = values[0] + values[5];
        values[9] =
        values[5] / 10;
        --values[2];

        for ( index = 0;  index < 10;  ++index )
                printf ("values[%i] = %i\n", index, values[index]);
}
```

Program 7.1 OUTPUT

```
values[0] = 197
values[1] = 0
values[2] = -101
values[3] = 547
values[4] = 0
values[5] = 350
values[6] = 0
values[7] = 0
values[8] = 0
values[9] = 35
```

The variable index assumes the values 0 through 9, as the last valid subscript of an array is always one less than the number of elements (due to that zeroth element). Since we never assigned values to five of the elements in the array—elements 1, 4, and 6 through 8—the values that are displayed for them are meaningless. Even though the program's output shows these values as zero, the value of any uninitialized variable or array element is simply the value that happens to be sitting around inside the computer's memory at the time that the program is executed. For this reason, no assumption should ever be made as to the value of an uninitialized variable or array element.

It is now time to consider a slightly more practical example. Suppose we took a telephone survey to discover how people felt about a particular television show and we asked each respondent to rate the show on a scale from 1 to 10, inclusive. After interviewing 5,000 people we accumulated a list of 5,000

numbers. Now we would like to analyze the results. One of the first pieces of data we would like to gather is a table showing the distribution of the ratings. In other words, we would like to know how many people rated the show a 1, how many a 2, and so on up to 10.

Although not an impossible chore, it would be a bit tedious to go through each response and manually count the number of responses in each rating category. And if we had a response which could be answered in more than ten ways (consider the task of categorizing the age of the respondent), this approach would be even more unreasonable. So we would like to develop a program to count the number of responses for each rating. The first impulse might be to set up ten different counters, called perhaps rating_1 through rating_10, and then to increment the appropriate counter each time the corresponding rating was keyed in. But once again, if we considered the case where we were dealing with more than ten possible choices, this approach could become a bit tedious. And besides, an approach that uses an array provides the vehicle for implementing a much cleaner solution, even in this case.

We can set up an array of counters called rating_counters, for example, and then we can increment the corresponding counter as each response is keyed in. Since we don't wish to take up 100 pages in this book for the 5,000 responses to the survey, in the program that follows we assume that we are dealing with only 20 responses. Anyway, it's always good practice to get a program working on a smaller test case first before proceeding with the full set of data, since problems that are discovered in the program will be much easier to isolate and debug if the amount of test data is small.

Program 7.2

```
#include <stdio.h>
main ()
{
        int  rating_counters[11], i, response;

        for ( i = 1;  i <= 10;  ++i )
                rating_counters[i] = 0;

        printf ("Enter your responses\n");

        for ( i = 1;  i <= 20;  ++i )
        {
                scanf ("%i", &response);
                if ( response < 1 || response > 10 )
                        printf ("Bad response: %i\n", response);
                else    ++rating_counters[response];
```

continues

Program 7.2 Continued

```
    }
    printf ("\n\nRating   Number of Responses\n");
    printf ("------ ----------------\n");

    for ( i = 1;  i <= 10;  ++i )
        printf ("%4d%14d\n", i, rating_counters[i]);
}
```

Program 7.2 OUTPUT

```
Enter your responses
6
5
8
3
9
6
5
7
15
Bad response: 15
5
5
1
7
4
10
5
5
8
8
9

Rating   Number of Responses
------ ----------------
   1             1
   2             0
   3             1
   4             1
   5             6
   6             3
   7             2
   8             2
   9             2
  10             1
```


The array `rating_counters` is defined to contain 11 elements. A valid question you might ask is, "If there are only 10 possible responses to the survey, why then is the array defined to contain 11 elements rather than 10?" The answer lies in the strategy for counting the responses in each particular rating category. Since each response can be a number from 1 to 10, the program keeps track of the responses for any one particular rating by simply incrementing the corresponding array element (after first checking to make sure that the user entered a valid response between 1 and 10). For example, if a rating of 5 is typed in, the value of `rating_counters[5]` is incremented by one. By employing this technique, the total number of respondents that rated the TV show a 5 will be contained in `rating_counters[5]`.

The reason for 11 elements versus 10 should now be clear. Since the highest rating number is a 10, we must set up our array to contain 11 elements in order to index `rating_counters[10]`, remembering that due to that zeroth element, the number of elements in an array is always one more than the highest index number. Since no response can have a value of zero, `rating_counters[0]` is never used. In fact, in the `for` loops that initialize and display the contents of the array, you will note that the variable `i` starts at 1, and thereby bypasses the initialization and display of `rating_counters[0]`.

As a point of discussion, it is mentioned that we could have developed our program to use an array containing precisely ten elements. Then, when each response was keyed in by the user, we could have instead incremented `rating_counters[response - 1]`. This way, `rating_counters[0]` would have contained the number of respondents that rated the show a 1, `rating_counters[1]` the number that rated the show a 2, and so on. This is a perfectly fine approach. The only reason it was not used was because storing the number of responses of value *n* inside `rating_counters[n]` is a slightly more straightforward approach.

Study Program 7.3, which generates a table of the first 15 *Fibonacci* numbers, and try to predict its output. What relationship exists between each number in the table?

Program 7.3

```
/* Program to generate the first 15 Fibonacci numbers */
#include <stdio.h>
main ()
{
    int  Fibonacci[15], i;

    Fibonacci[0] = 0;    /* by definition */
    Fibonacci[1] = 1;    /*     ditto     */
```

continues

Program 7.3 Continued

```
    for ( i = 2;  i < 15;  ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];
    for ( i = 0;  i < 15;  ++i )
        printf ("%i\n", Fibonacci[i]);
}
```

Program 7.3 OUTPUT

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

The first two Fibonacci numbers, which we will call F_0 and F_1, are defined to be 0 and 1, respectively. Thereafter, each successive Fibonacci number F_i is defined to be the sum of the two preceding Fibonacci numbers F_{i-2} and F_{i-1}. So F_2 is calculated by adding together the values of F_0 and F_1. In the preceding program, this corresponds directly to calculating `Fibonacci[2]` by adding the values `Fibonacci[0]` and `Fibonacci[1]`. This calculation is performed inside the `for` loop, which calculates the values of F_2 through F_{14} (or, equivalently, `Fibonacci[2]` through `Fibonacci[14]`).

Fibonacci numbers actually have many applications in the field of mathematics and in the study of computer algorithms. The sequence of Fibonacci numbers historically originated from the "rabbits problem": If we start with a pair of rabbits and assume that each pair of rabbits produces a new pair of rabbits each month, that each newly born pair of rabbits can produce offspring by the end of their second month, and that rabbits never die, how many pairs of rabbits will there be after the end of a year? The answer to this problem rests in the fact that at the end of the nth month, there will be a total of F_{n+2} rabbits.

Therefore, according to the table from Program 7.3's output, at the end of the twelfth month, there will be a total of 377 pairs of rabbits.

Now it's time to return to the prime number program that we developed in Chapter 6, "Making Decisions," and see how the use of an array can help us to develop a more efficient program. In Program 6.10, the criteria that we used for determining if a number was prime was to divide the prime candidate by all successive integers from two up to the number minus one. In Exercise 7 in Chapter 6, we noted two inefficiencies with this approach that could easily be corrected. But even with these changes, the approach used is still not efficient. And while such questions of efficiency may not be important when dealing with a table of prime numbers up to 50, these questions do become important, for example, when we start thinking about generating a table of prime numbers up to 100,000.

One method for generating prime numbers that is an improvement over the previous approach involves the notion that a number is prime if it is not evenly divisible by any other prime number. This stems from the fact that any nonprime integer can be expressed as a multiple of prime factors. (For example, 20 has the prime factors 2, 2, and 5.) We can use this added insight to help us to develop a more efficient prime number program. The program can test if a given integer is prime by determining if it is evenly divisible by any other previously generated prime. By now the term "previously generated" should trigger off in your mind the idea that an array must be involved here. We can use an array to store each prime number as it is generated.

As a further optimization of the prime number generator program, it can be readily demonstrated that any nonprime integer n must have as one of its factors an integer that is less than or equal to the square root of n. What this means is that it is only necessary to determine if a given integer is prime by testing it for even divisibility against all prime factors up to the square root of the integer.

Program 7.4 incorporates the above discussions into a program to generate all prime numbers up to 50. The expression

```
p / primes[i] >= primes[i]
```

is used in the innermost `for` loop as a test to ensure that the value of `primes[i]` does not exceed the square root of `p`. This test comes directly from the discussions in the last paragraph. (You might want to think about the math a bit.)

We start off by storing 2 and 3 as the first two primes into the array `primes`. This array has been defined to contain 50 elements, even though we obviously won't need that many locations for storing the prime numbers. The variable `prime_index` is initially set to 2, which is the next free slot in the `primes` array. A `for` loop is then set up to run through the odd integers from 5 to 50. After the

flag is_prime is set to TRUE, another for loop is entered. This loop will successively divide the value of p by all of the previously generated prime numbers that are stored in the array primes. The index variable i starts at 1, since it is not necessary to test any values of p for divisibility by primes[0] (which is 2). This is true because our program does not even consider even numbers as possible primes. Inside the loop, a test is made to see if the value of p is evenly divisible by primes[i], and if it is, then is_prime is set FALSE. The for loop continues execution so long as the value of is_prime is TRUE *and* the value of primes[i] does not exceed the square root of p.

After exiting the for loop, a test of the is_prime flag determines whether or not to store the value of p as the next prime number inside the primes array.

Once all values of p have been tried, the program displays each prime number that has been stored inside the primes array. The value of the index variable i varies from 0 through prime_index - 1, since prime_index was always set pointing to the *next* free slot in the primes array.

Program 7.4

```
#include <stdio.h>
/* Modified program to generate prime numbers */

main ()
{
      int  p, is_prime, i, primes[50], prime_index = 2;

      primes[0] = 2;
      primes[1] = 3;

      for ( p = 5;  p <= 50;  p = p + 2 )
      {
         is_prime = 1;

         for ( i = 1;  is_prime  &&
                       p / primes[i] >= primes[i]; ++i )
            if ( p % primes[i] == 0 )
               is_prime = 0;

         if ( is_prime )
         {
            primes[prime_index] = p;
            ++prime_index;
         }
   }
```



```
      for ( i = 0;  i < prime_index;  ++i )
         printf ("%i  ", primes[i]);

      printf ("\n");
}
```

Program 7.4 OUTPUT

```
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47
```

Initializing Array Elements

Just as we can assign initial values to variables when they are declared, so can we assign initial values to the elements of an array. This is done by simply listing the initial values of the array, starting from the first element. Values in the list are separated by commas and the entire list is enclosed in a pair of braces.

The statement

```
int  counters[5] = { 0, 0, 0, 0, 0 };
```

will declare an array called counters to contain 5 integer values and will initialize each of these elements to zero. In a similar fashion, the statement

```
int  integers[5] = { 0, 1, 2, 3, 4 };
```

will set the value of integers[0] to 0, integers[1] to 1, integers[2] to 2, and so on.

Arrays of characters are initialized in a similar manner; thus the statement

```
char  letters[5] = { 'a', 'b', 'c', 'd', 'e' };
```

will define the character array letters and will initialize the five elements to the characters 'a', 'b', 'c', 'd', and 'e', respectively.

It is not necessary to completely initialize an entire array. If fewer initial values are specified, only an equal number of elements will be initialized. The remaining values in the array will be set to zero. So the declaration

```
float  sample_data[500] = { 100.0, 300.0, 500.5 };
```

initializes the first three values of sample_data to 100.0, 300.0, and 500.5, and sets the remaining 497 elements to zero.

Unfortunately, C does not provide any shortcut mechanisms for initializing array elements, such as the type provided in FORTRAN, for example. There is no way to specify a repeat count, so if it were desired to initially set all 500 values of `sample_data` to 1, then all 500 would have to be explicitly spelled out. In such a case, it would be better to initialize the array inside the program using an appropriate `for` loop.

Program 7.5 illustrates the two types of array-initialization techniques.

Program 7.5

```
#include <stdio.h>
main ()
{
   int  array_values[10] = { 0, 1, 4, 9, 16 };
   int  i;

   for ( i = 5;  i < 10;  ++i )
      array_values[i] = i * i;

   for ( i = 0;  i < 10;  ++i )
      printf ("array_values[%i] = %i\n", i, array_values[i]);
}
```

Program 7.5 OUTPUT

```
array_values[0] = 0
array_values[1] = 1
array_values[2] = 4
array_values[3] = 9
array_values[4] = 16
array_values[5] = 25
array_values[6] = 36
array_values[7] = 49
array_values[8] = 64
array_values[9] = 81
```

In the declaration of the array `array_values`, the first five elements of the array are initialized to the square of their element number (for example, element number 3 is set equal to 3_2 or 9). The first `for` loop shows how this same type of initialization can be performed inside a loop. This loop sets each of the elements 5 through 9 to the square of its element number. The second `for` loop simply runs through all ten elements to display their values at the terminal.

Character Arrays

The purpose of Program 7.6 is to simply illustrate how a character array can be used. However, there is one point worthy of discussion. Can you spot it?

Program 7.6

```
#include <stdio.h>
main ()
{
   char  word[] = { 'H', 'e', 'l', 'l', 'o', '!' };
   int   i;

   for ( i = 0;  i < 6;  ++i )
     printf ("%c", word[i]);

   printf ("\n");
}
```

Program 7.6 OUTPUT

```
Hello!
```

The most notable point in the preceding program is the declaration of the character array `word`. There is no mention of the number of elements in the array. The C language allows you to define an array without specifying the number of elements. If this is done, then the size of the array will be determined automatically based on the number of initialization elements. Since in Program 7.6 there are six initial values listed for the array `word`, the C language implicitly dimensions the array to six elements.

This approach works fine so long as you initialize every element in the array at the point that the array is defined. If this is not to be the case, then you must explicitly dimension the array.

The next program further illustrates the use of integer and character arrays. The task is to develop a program that converts a positive integer from its base- 10 representation into its equivalent representation in another base up to base- 16. As inputs to the program, we will specify the number to be converted and also the base to which we would like the number converted to. The program will then convert the keyed-in number to the appropriate base and display the result.

The first step in developing such a program is to devise an algorithm to convert a number from base 10 to another base. An algorithm to generate the digits of the converted number can be informally stated as follows: A digit of the converted number is obtained by taking the modulo of the number by the base. The number is then divided by the base, with any fractional remainder discarded, and the process is repeated until the number reaches 0.

The outlined procedure will generate the digits of the converted number starting from the rightmost digit. Why don't we pick an example and see how it works? Suppose we wanted to convert the number 10 into base 2. The following table shows the steps that would be followed to arrive at the result.

Number	Number Modulo 2	Number / 2
10	0	5
5	1	2
2	0	1
1	1	0

The result of converting 10 to base 2 is therefore seen to be 1010, reading the digits of the "Number Modulo 2" column from the bottom to the top.

In order to write a program that performs the preceding conversion process, we must take a couple of things into account. First of all, the fact that the algorithm generates the digits of the converted number in reverse order is not very nice. We certainly don't expect the user to read the result from right to left, or from the bottom of the page upward. Therefore, we must correct this problem. Rather than simply displaying each digit as it is generated, we can have the program store each digit inside an array. Then, when we have finished converting the number, we can display the contents of the array in the correct order.

The second thing that must be realized is that we specified that the program handle conversion of numbers into bases up to 16. This means that any digits of the converted number that are between 10 and 15 must be displayed using the corresponding letters, A through F. This is where our character array enters the picture.

Examine Program 7.7 to see how these two issues are handled.

The character array `base_digits` is set up to contain the 16 possible digits that will be displayed for the converted number. The array `converted_number` is defined to contain a maximum of 64 digits, which will hold the results of converting the largest possible long integer to the smallest possible base (base 2) on just about all machines. We defined the variable `number_to_convert` to be of

type `long int` so that relatively large numbers can be converted if desired. Finally, the variables `base` (to contain the desired conversion base) and `index` (to index into the `converted_number` array) are both defined to be of type `int`.

After the user keys in the values of the number to be converted and the base—and you will note that the `scanf` call to read in a long integer value takes the format characters `%ld`—the program then enters a `do` loop to perform the conversion. The `do` was chosen so that at least one digit will appear in the `converted_number` array even if the user keys in the number 0 to be converted.

Inside the loop, the `number_to_be_converted` modulo the `base` is computed to determine the next digit. This digit is stored inside the `converted_number` array, and the `index` into the array incremented by 1. After dividing the `number_to_be_converted` by the base, the conditions of the `do` are checked. If the value of `number_to_be_converted` is 0, the loop terminates; otherwise, the loop is repeated to determine the next digit of the converted number.

Program 7.7

```
/* Program to convert a positive integer to another base */
#include <stdio.h>
main ()
{
     char  base_digits[16] =
                    { '0', '1', '2', '3', '4', '5', '6', '7',
                      '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
     int converted_number[64];
     long int number_to_convert;
     int next_digit, base, index = 0;

     /* get the number and the base */

     printf ("Number to be converted? ");
     scanf ("%ld", &number_to_convert);
     printf ("Base? ");
     scanf ("%i", &base);

     /* convert to the indicated base */

     do
     {
        converted_number[index] = number_to_convert % base;
        ++index;
        number_to_convert = number_to_convert / base;
     }
     while  ( number_to_convert != 0 );
```

continues

Program 7.7 Continued

```
/* display the results in reverse order */
printf ("Converted number = ");
for ( index;  index >= 0;  --index )
{
        next_digit = converted_number[index];
        printf ("%c", base_digits[next_digit]);
}
printf ("\n");
}
```

Program 7.7 OUTPUT

```
Number to be converted? 10
Base? 2
Converted number = 1010
```

Program 7.7 OUTPUT (Rerun)

```
Number to be converted? 128362
Base? 16
Converted number = 1F56A
```

When the `do` loop is done, the value of the variable `index` will be the number of digits in the converted number. Since this variable will be incremented one time too many inside the `do` loop, its value is initially decremented by 1 in the `for` loop. The purpose of this `for` loop is to display the converted number at the terminal. The `for` loop sequences through the `converted_number` array in *reverse* sequence to display the digits in the correct order.

Each digit from the `converted_number` array is in turn assigned to the variable `next_digit`. In order that the numbers 10 through 15 be correctly displayed using the letters A through F, a lookup is then made inside the array `base_digits`, using the value of `next_digit` as the index. For the digits 0 through 9, the corresponding location in the array `base_digits` contains nothing more than the characters `'0'` through `'9'` (which as you will recall *are* distinct from the integers 0 through 9). Locations 10 through 15 of the array contain the characters `'A'` through `'F'`. So if the value of `next_digit` is 10, for example, then the character contained in `base_digits[10]`, or `'A'`, will be displayed. And if the value of `next_digit` is 8, then the character `'8'` as contained in `base_digits[8]` will be displayed.

When the value of `index` becomes less than 0, the `for` loop will be finished. At that point the program will display a newline character, and program execution will be terminated.

Incidentally, you might be interested in knowing that you could have easily avoided the intermediate step of assigning the value of `converted_number[index]` to `next_digit` by directly specifying this expression as the subscript of the `base_digits` array in the `printf` call. In other words, the expression

```
base_digits[ converted_number[index] ]
```

could have been supplied to the `printf` routine and the same results achieved. Of course, this expression is a bit more cryptic than the two equivalent expressions used by the program.

It should be pointed out that we were a bit sloppy in the preceding program. No check was ever made to ensure that the value of `base` was between 2 and 16. If the user had entered 0 for the value of the base, the division inside the `do` loop would have been a division by 0. That's something we should never let happen. And if the user had keyed in 1 as the value of the base, then the program would go into an infinite loop since the value of `number_to_convert` would never reach 0. If the user entered a base value that was greater than 16, there is a chance that we would have exceeded the bounds of the `base_digits` array later in the program. That's another "gotcha" that we must avoid, since the C system does not check this condition for us.

In the next chapter, "Functions," we will rewrite this program and resolve these issues. But now let's take a look at an interesting extension to the notion of an array.

Multidimensional Arrays

The types of arrays that we have been exposed to so far are all linear arrays—that is, they all dealt with a single dimension. The C language allows arrays of any dimension to be defined. In this section, we will take a look at two-dimensional arrays.

One of the most natural applications for a two-dimensional array arises in the case of a matrix. Consider the 4 x 5 matrix shown next.

```
10  5 -3  17  82
 9  0  0   8  -7
32 20  1   0  14
 0  0  8   7   6
```

In mathematics, it is quite common to refer to an element of a matrix by use of a double subscript. So if we called the preceding matrix M, then the notation M_{ij} would refer to the element in the ith row , jth column, where i ranges from 1 through 4, and j ranges from 1 through 5. The notation $M_{3,2}$ would refer to the value 20, which is found in the 3rd row, 2nd column of the matrix. In a similar fashion, $M_{4,5}$ would refer to the element contained in the 4th row, 5th column: the value 6.

In C, there is an analogous notation to be used when referring to elements of a two-dimensional array. However, since C likes to start numbering things at 0, the first row of the matrix is actually row 0, and the first column of the matrix is column 0. The preceding matrix would then have row and column designations as shown in the diagram below.

Row (i)	Column (j)				
	0	1	2	3	4
0	10	5	-3	17	82
1	9	0	0	8	-7
2	32	20	1	0	14
3	0	0	8	7	6

Whereas in mathematics the notation $M_{i,j}$ is used, in C the equivalent notation is

```
M[i][j]
```

Remember, the first index number refers to the row number, while the second index number references the column. So the statement

```
sum = M[0][2] + M[2][4];
```

would add the value contained in row 0, column 2—which is -3—to the value contained in row 2, column 4—which is 14—and would assign the result of 11 to the variable sum.

Two-dimensional arrays are declared the same way that one-dimensional arrays are; thus

```
int  M[4][5];
```


declares the array M to be a two-dimensional array consisting of 4 rows and 5 columns, for a total of 20 elements. Each position in the array is defined to contain an integer value.

Two-dimensional arrays may be initialized in a manner analogous to their one-dimensional counterparts. When listing elements for initialization, the values are listed by row. Brace pairs are used to separate the list of initializers for one row from the next. So to define and initialize the array M to the elements listed in the preceding table, a statement such as the following could be used.

```
int  M[4][5] = {
                 { 10,  5, -3, 17, 82 },
                 {  9,  0,  0,  8, -7 },
                 { 32, 20,  1,  0, 14 },
                 {  0,  0,  8,  7,  6 }
               };
```

Pay particular attention to the syntax of the above statement. Note that commas are required after each brace that closes off a row, except in the case of the last row. The use of the inner pairs of braces is actually optional. If not supplied, then initialization proceeds by row. Thus the above statement could also have been written as

```
int  M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, -7, 32,
                 20, 1, 0, 14, 0, 0, 8, 7, 6 };
```

As with one-dimensional arrays, it is not required that the entire array be initialized. A statement such as

```
int  M[4][5] = {
                 { 10,  5, -3 },
                 {  9,  0,  0 },
                 { 32, 20,  1 },
                 {  0,  0,  8 }
               };
```

would only initialize the first three elements of each row of the matrix to the indicated values. The remaining values will be set to 0. Note that, in this case, the inner pairs of braces *are required* to force the correct initialization. Without them, the first two rows and the first two elements of the third row would have been initialized instead. (Verify to yourself that this would be the case.)

Exercises

A program example showing the use of multidimensional arrays is deferred to the next chapter, where we will begin our detailed discussion of one of the most important concepts in the C language—the program *function*. Before proceeding to that chapter, however, try to work the exercises that follow.

1. If you have access to a computer facility that supports the C programming language, type in and run the seven programs presented in this chapter. Compare the output produced by each program with the output presented after each program.

2. Modify Program 7.1 so that the elements of the array `values` are initially set to 0. Use a `for` loop to perform the initialization.

3. Program 7.2 permits only 20 responses to be entered. Modify that program so that any number of responses may be keyed in. So that the user does not have to count the number of responses in the list, set up the program so that the value 999 can be keyed in by the user to indicate that the last response has been entered. (Hint: you can use the `break` statement here if you want to exit your loop.)

4. Write a program that calculates the average of an array of ten floating-point values.

5. What output would you expect from the following program?

```
#include <stdio.h>
main ()
{
    int  numbers[10] =
            { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    int  i, j;

    for ( j = 0;  j < 10;  ++j )
        for ( i = 0;  i < j;  ++i )
            numbers[j]  =  numbers[j] + numbers[i];
    for ( j = 0;  j < 10;  ++j )
        printf ("%i ", numbers[j]);

    printf ("\n");
}
```

6. Prime numbers may also be generated by an algorithm known as the *Sieve of Erastosthenes*. The algorithm for this procedure is presented here. Write a program that implements this algorithm. Have the program find all prime numbers up to 150. What can you say about this algorithm as compared to the ones used in the text for calculating prime numbers?

Sieve of Erastosthenes Algorithm
To Display All Prime Numbers Between 1 and n

Step 1:	Define an array of integers P. Set all elements P_i to 0, $2 <= i <= n$.
Step 2:	Set i to 2.
Step 3:	If $i > n$, the algorithm terminates.
Step 4:	If P_i is zero, then i is prime.
Step 5:	For all positive integer values of j, such that $i \times j < n$, set P_{ixj} to 1.
Step 6:	Add 1 to i and go to Step 3.

8

CHAPTER

Functions

Behind all well-written programs in the C programming language lies the same fundamental element—the *function*. We have used functions in every program that we have encountered thus far. The `printf` and `scanf` routine are examples of functions. Indeed, each and every program also used a function called `main`. So you may ask, what is all the fuss about? The truth is that the program function provides the mechanism for producing programs that are easy to write, read, understand, debug, modify, and maintain. Obviously, anything that can accomplish all of these things is worthy of a bit of fanfare.

Let us first understand what a function is, and then proceed to show how it can be most effectively used in the development of programs. Let's go back to the very first program that we wrote (Program 3.1), which displayed the phrase "Programming is fun" at the terminal:

```
#include <stdio.h>
main ()
{
    printf ("Programming is fun.\n");
}
```

Here is a function called `print_message` that does the same thing:

```
void print_message (void)
{
    printf ("Programming is fun.\n");
}
```

The only difference between `print_message` and the function `main` from Program 3.1 is in the first line. The first line of a function definition tells the compiler four things about the function:

1. Its name.
2. The *type* of value it *returns*.
3. The arguments it takes.
4. Who can call it (discussed in Chapter 15, "Working with Larger Programs").

The first line of the `print_message` function definition tells the compiler that `print_message` is the name of the function, that it returns no value (the first use of the keyword `void`), and that it takes no arguments (the second use of the keyword `void`). We'll go into more details about the `void` keyword shortly.

> **QUICK TIP**
>
> Choosing meaningful function names is just as important as choosing meaningful variable names—the choice of names greatly affects the program's readability.

In our discussions of Program 3.1 we mentioned that `main` was a specially recognized name in the C system that always indicated where the program was to begin execution. There *always* must be a `main`. We can add a `main` function to the preceding code to end up with a complete program, as shown next.

Program 8.1

```
#include <stdio.h>
void print_message (void)
{
    printf ("Programming is fun.\n");
}

main ()
{
    print_message ();
}
```


Program 8.1 OUTPUT

```
Programming is fun.
```

Program 8.1 consists of *two* functions: `print_message` and `main`. Program execution always begins with `main`. Inside that function, the statement

```
print_message ();
```

appears. This statement indicates that the function `print_message` is to be executed. The open and closed parentheses are used to tell the compiler that `print_message` is a function and that no arguments or values are to be passed to this function (which is consistent with the way we defined the function in the program). When a function call is executed, program execution is transferred directly to the indicated function. Inside the `print_message` function, the `printf` statement will be executed to display the message "Programming is fun" at the terminal. After the message has been displayed, the `print_message` routine will be finished (as signalled by the closed brace) and the program will *return* to the `main` routine, where program execution will continue at the point where the function call was executed.

As mentioned previously, the idea of calling a function is not new. The `printf` and `scanf` routines are both program functions. The main distinction here is that these routines did not have to be written by us since they are a part of the C program library. Whenever we used the `printf` function to display a message or program results, execution was transferred to the `printf` function, which performed the required tasks and then returned back to the program. In each case, execution was returned to the program statement that immediately followed the call to the function.

Now try to predict the output from the following program.

Program 8.2

```
#include <stdio.h>
void print_message (void)
{
     printf ("Programming is fun.\n");
}

main ()
{
    print_message ();
    print_message ();
}
```

Program 8.2 OUTPUT

```
Programming is fun.
Programming is fun.
```

Execution of the preceding program starts at `main`, which contains two calls to the `print_message` function. When the first call to the function is executed, control is sent directly to the `print_message` function, which displays the message "Programming is fun" at the terminal and then returns to the `main` routine. Upon return, another call to the `print_message` routine is encountered, which results in the execution of the same function a second time. After the return is made from the `print_message` function, execution is terminated.

As a final example of the `print_message` function, try to guess the output from the following program.

Program 8.3

```
#include <stdio.h>
void print_message (void)
{
    printf ("Programming is fun.\n");
}

main ()
{
    int  i;

    for ( i = 1;  i <= 5;  ++i )
        print_message ();
}
```

Program 8.3 OUTPUT

```
Programming is fun.
Programming is fun.
Programming is fun.
Programming is fun.
Programming is fun.
```


Arguments and Local Variables

When the `printf` function is called, we always supply one or more values to the function, the first value being the format string and the remaining values the specific program results to be displayed. These values, called *arguments*, greatly increase the usefulness and flexibility of a function. Unlike our `print_message` routine, which will display the same message each time it is called, the `printf` function will display whatever you tell it to display.

We can define a function that accepts arguments. In Chapter 5, "Program Looping," we developed an assortment of programs for calculating triangular numbers. Here we'll define a function to generate a triangular number. We'll call it, appropriately enough, `calculate_triangular_number`. As an argument to the function, we will specify which triangular number to calculate. The function will then calculate the desired number and display the results at the terminal. Here then is the function to accomplish the task and a `main` routine to try it out.

Program 8.4

```
/* Function to calculate the nth triangular number */
#include <stdio.h>
void calculate_triangular_number (int n)
{
    int  i, triangular_number = 0;

    for ( i = 1;  i <= n;  ++i )
        triangular_number = triangular_number + i;

    printf ("Triangular number %i is %i\n", n, triangular_number);
}

main ()
{
    calculate_triangular_number (10);
    calculate_triangular_number (20);
    calculate_triangular_number (50);
}
```

Program 8.4 OUTPUT

```
Triangular number 10 is 55
Triangular number 20 is 210
Triangular number 50 is 1275
```

The function `calculate_triangular_number` requires a bit of explanation. The first line of the function:

```
void calculate_triangular_number (int n)
```

is called the *function prototype declaration*. It tells the compiler that `calculate_triangular_number` is a function that returns no value (the keyword `void`) and that takes a single argument, called `n`, which is an `int`. The name that is chosen for an argument, called its *formal parameter name*, as well as the name of the function itself, can be any valid name formed by observing the rules outlined in Chapter 4, "Variables, Data Types, and Arthmetic Expressions," for forming variable names. For obvious reasons, you should choose meaningful names.

Once the formal parameter name has been defined, it can be used to refer to the argument anywhere inside the body of the function.

The beginning of the function's definition is indicated by the opening curly brace. Since we wish to calculate the nth triangular number, we have to set up a variable to store the value of the triangular number as it is being calculated. We also need a variable to act as our loop index. The variables `triangular_number` and `i` are defined for these purposes and are declared to be of type `int`. These variables are defined and initialized in the same manner that we defined and initialized our variables inside the `main` routine in previous programs.

> **QUICK TIP**
>
> If an initial value is given to a variable inside a function, that initial value will be assigned to the variable *each* time the function is called.

Variables defined inside a function are known as *automatic local* variables, since they are automatically "created" each time the function is called, and since their values are local to the function.

> **QUICK TIP**
>
> The value of a local variable can only be accessed by the function in which the variable is defined. Its value cannot be accessed by any other function.

When defining a local variable inside a function, it is more precise in C to use the keyword `auto` before the definition of the variable. An example of this would be the following:


```
auto int  i, triangular_number = 0;
```

Since the C compiler assumes by default that any variable defined inside a function is an automatic local variable, the keyword `auto` is seldom used, and for this reason it will not be used in this book.

Returning to our program example, after the local variables have been defined, the function calculates the triangular number and displays the results at the terminal. The closed brace then defines the end of the function.

Inside the `main` routine, the value 10 is passed as the argument in the first call to `calculate_triangular_number`. Execution is then transferred directly to the function where the value 10 *becomes the value of the formal parameter* `n` inside the function. The function then proceeds to calculate the value of the tenth triangular number and display the result.

The next time that `calculate_triangular_number` is called, the argument 20 is passed. In a similar process, as described earlier, this value becomes the value of `n` inside the function. The function then proceeds to calculate the value of the twentieth triangular number and display the answer at the terminal.

For an example of a function that takes more than one argument, let us rewrite the greatest common divisor program (Program 5.7) in function form. The two arguments to the function will be the two numbers whose greatest common divisor (*gcd*) we wish to calculate.

Program 8.5

```
/* This function finds the greatest common divisor
      of two nonnegative integer values                */
#include <stdio.h>

void gcd (int u, int v)
{
    int  temp;

    printf ("The gcd of %i and %i is ", u, v);

    while ( v != 0 )
    {
        temp = u % v;
        u = v;
        v = temp;
    }

    printf ("%i\n", u);
}
```

continues

Program 8.5 Continued

```
main ()
{
    gcd (150, 35);
    gcd (1029, 405);
    gcd (83, 240);
}
```

Program 8.5 OUTPUT

```
The gcd of 150 and 35 is 5
The gcd of 1029 and 405 is 27
The gcd of 83 and 240 is 1
```

The function `gcd` is defined to take two integer arguments. The function will refer to these arguments through their formal parameter names `u` and `v`. After declaring the variable `temp` to be of type `int`, the program displays the values of the arguments `u` and `v`, together with an appropriate message at the terminal. The function then calculates and displays the greatest common divisor of the two integers.

You may be wondering why we have two `printf` statements inside the function `gcd`. We must display the values of `u` and `v` *before* we enter the `while` loop, since their values are changed inside the loop. If we waited until after the loop had finished, the values displayed for `u` and `v` would not at all resemble the original values that were passed to the routine. Another solution to this problem would have been to assign the values of `u` and `v` to two variables before entering the `while` loop. The values of these two variables could have then been displayed together with the value of `u` (the greatest common divisor) using a single `printf` statement after the `while` loop was completed.

Returning Function Results

The functions in Programs 8.4 and 8.5 perform some straightforward calculations and then display the results of the calculations at the terminal. However, we may not always wish to have the results of our calculations displayed. The C language provides us with a convenient mechanism whereby the results of a function may be returned to the calling routine. The syntax of this construct is straightforward enough:

```
return (expression);
```


This statement indicates that the function is to return the value of `expression` to the calling routine. The parentheses around `expression` are actually optional, but seem to be used by many programmers.

However, an appropriate `return` statement is not enough. When the function declaration is made, we must also declare the *type of value the function returns*. This declaration is placed immediately *before* the function's name. Each of the previous examples in this chapter defined functions that did not return a value, that's why we placed the keyword `void` directly before the function name. On the other hand, a function declaration that started like this:

```
float  kmh_to_mph (float  km_speed)
```

would begin the definition of a function `kmh_to_mph`, which takes one `float` argument called `km_speed` and which *returns* a floating-point value. Similarly,

```
int  gcd (int  u, int  v)
```

defines a function `gcd` with integer arguments `u` and `v` that returns an integer value. In fact, let us modify Program 8.5 so that the greatest common divisor is not displayed by the function `gcd` but is instead returned to the `main` routine.

Program 8.6

```
/* This function finds the greatest common divisor of two
    nonnegative integer values and returns the result   */
#include <stdio.h>
int  gcd (int u, int v)
{
    int  temp;

    while ( v != 0 )
    {
        temp = u % v;
        u = v;
        v = temp;
    }

    return (u);
}

main ()
{
    int  result;

    result = gcd (150, 35);
    printf ("The gcd of 150 and 35 is %i\n", result);

    result = gcd (1029, 405);
```

continues

Program 8.6 Continued

```
    printf ("The gcd of 1026 and 405 is %i\n", result);

    printf ("The gcd of 83 and 240 is %i\n", gcd (83, 240));
}
```

Program 8.6 OUTPUT

```
The gcd of 150 and 35 is 5
The gcd of 1026 and 405 is 27
The gcd of 83 and 240 is 1
```

Once the value of the greatest common divisor has been calculated by the `gcd` function, the statement

```
return (u);
```

is executed. This has the effect of returning the value of `u`, which is the value of the greatest common divisor, back to the calling routine.

You might be wondering what we can do with the value that is returned to the calling routine. As you can see from the `main` routine, in the first two cases, the value that is returned is stored in the variable `result`. More precisely, the statement

```
result = gcd (150, 35);
```

says to call the function `gcd` with the arguments 150 and 35 and to store the value that is returned by this function into the variable `result`.

The result that is returned by a function does not have to be assigned to a variable, as you will observe by the last statement in the `main` routine. In this case, the result returned by the call

```
gcd (83, 240)
```

is passed directly to the `printf` function, where its value is displayed.

A C function can only return a single value in the manner that we have just described. Unlike FORTRAN or Pascal, C makes no distinction between subroutines (procedures) and functions. In C, there is only the function, which can optionally return a value. If the declaration of the type returned by a function is omitted, the C compiler assumes that the function will return an integer—if it returns a value at all. Many C programmers take advantage of this fact and

omit the return type declaration on functions that return integers. This is a bad programming habit that should be avoided.

> **QUICK TIP**
>
> Whenever a function returns a value, make sure you declare the type of value returned in the function's header, if only for the sake of improving the program's readability. In this manner, you will always be able to tell from the function header not only the function's name and the number and type of its arguments, but also if it returns a value and what the type of the returned value is.

As noted earlier, a function declaration that is preceded by the keyword `void` explicitly informs the compiler that the function does not return a value. A subsequent attempt at using the function in an expression, as if a value were returned, will result in a compiler error message. For example, since the `calculate_triangular_number` function of Program 8.4 did not return a value, we placed the keyword `void` before its name when defining the function. Subsequently attempting to use this function as if it returned a value, as in

```
number = calculate_triangular_number (20);
```

would result in a compiler error.

In a sense, the `void` data type is actually defining the *absence* of a data type. Therefore, a function declared to be of type `void` has no value and cannot be used as if it does in an expression.

In Chapter 6, "Making Decisions," we wrote a program to calculate and display the absolute value of a number. Let us now write a function that takes the absolute value of its argument and then returns the result. Instead of using integer values as we did in Program 6.1, we'll write this function to take a floating value as an argument and to also return the answer as type `float`.

Program 8.7

```
/* Function to calculate the absolute value */
#include <stdio.h>
float  absolute_value (float x)
{
    if ( x < 0 )
      x = -x;
```

continues

Program 8.7 Continued

```
    return (x);
}

main ()
{
    float  f1 = -15.5, f2 = 20.0, f3 = -5.0;
    int    i1 = -716;
    float  result;

    result = absolute_value (f1);
    printf ("result = %.2f\n", result);
    printf ("f1 = %.2f\n", f1);

    result = absolute_value (f2) + absolute_value (f3);
    printf ("result = %.2f\n", result);

    result = absolute_value ( (float) i1 );
    printf ("result = %.2f\n", result);

    result = absolute_value (i1);
    printf ("result = %.2f\n", result);

    printf ("%.2f\n", absolute_value (-6.0) / 4 );
}
```

Program 8.7 OUTPUT

```
result = 15.50
f1 = -15.50
result = 25.00
result = 716.00
result = 716.00
1.50
```

The `absolute_value` function is relatively straightforward. The formal parameter called `x` is tested against 0. If it is less than 0, the value is negated to take its absolute value. The result is then returned back to the calling routine with an appropriate `return` statement.

There are some interesting points with respect to the `main` routine that tests out the `absolute_value` function. In the first call to the function, the value of the variable `f1`, initially set to -15.5, is passed. Inside the function itself, this value is assigned to the variable `x`. Since the result of the `if` test will be TRUE, the

statement that negates the value of `x` will be executed, thereby setting the value of `x` to 15.5. In the next statement, the value of `x` is returned to the `main` routine where it is assigned to the variable `result` and then displayed.

When the value of `x` is changed inside the `absolute_value` function, this in no way affects the value of the variable `f1`. When `f1` was passed to the `absolute_value` function, its *value was automatically copied* into the formal parameter `x` by the system. Therefore, any changes made to the value of `x` inside the function affect only the value of `x` and not the value of `f1`. This is verified by the second `printf` call, which displays the unchanged value of `f1` at the terminal.

> **QUICK TIP**
>
> It's not possible for a function to directly change the value of any of its arguments—it can only change copies of them.

The next two calls to the `absolute_value` function illustrate how the result returned by a function can be used in an arithmetic expression. The absolute value of `f2` is added to the absolute value of `f3` and the sum is assigned to the variable `result`.

The fourth call to the `absolute_value` function introduces the notion that the type of argument that is passed to a function should agree with the type of the argument as declared inside the function. Since the function `absolute_value` expects a floating value as its argument, we first cast our integer variable `i1` to type `float` before the call is made. If you omit the cast operation, the compiler will do it for you anyway, since it knows the `absolute_value` function is expecting a floating argument. (This is verified by the fifth call to the `absolute_value` function.) However, it's clearer what's going on if you do the casting yourself rather than relying on the compiler to do the conversion for you.

The final call to the `absolute_value` function shows that the rules for evaluation of arithmetic expressions also pertain to values returned by functions. Since the value returned by the `absolute_value` function is declared to be of type `float`, the compiler treats the division operation as the division of a floating-point number by an integer. As you will recall, if one operand of a term is of type `float`, then the operation is performed using floating arithmetic. In accordance with this rule, the division of the absolute value of -6.0 by 4 produces a result of 1.5.

Now that we have defined a function that computes the absolute value of a number, we can use it in any future programs where we might need such a calculation performed. In fact, the next program is just such an example.

Functions Calling Functions Calling ...

With today's pocket calculators as commonplace as wristwatches, it is usually no big deal to find the square root of a particular number should the need arise. But years ago, students were taught manual techniques that could be used to arrive at an approximation of the square root of a number. One such approximation method that lends itself most readily to solution by a computer is known as the *Newton-Raphson Iteration Technique*. In the next program, we'll write a square root function that uses this technique to arrive at an approximation of the square root of a number.

The Newton-Raphson method may be easily described as follows. We begin by selecting a "guess" at the square root of the number. The closer that this guess is to the actual square root, the fewer the number of calculations that will have to be performed to arrive at the square root. For the sake of argument, however, we will assume that we are not very good at guessing and will therefore always make an initial guess of 1.

The number whose square root we wish to obtain is divided by the initial guess and is then added to the value of guess. This intermediate result is then divided by 2. The result of this division becomes the new guess for another go-around with the formula. That is, the number whose square root we are calculating is divided by this new guess, added into this new guess, and then divided by 2. This result then becomes the new guess and another iteration is performed.

Since we don't wish to continue this iterative process forever, we need some way of knowing when to stop. Since the successive guesses that are derived by repeated evaluation of the formula get closer and closer to the true value of the square root, we can set a limit that we can use for deciding when to terminate the process. The difference between the square of the guess and the number itself can then be compared against this limit—usually called epsilon (ε). If the difference is less than ε, then the desired accuracy for the square root will have been obtained and the iterative process can be terminated.

This procedure can be expressed in terms of an algorithm as shown.

Newton-Raphson Method to Compute the Square Root of x

Step 1:	Set the value of guess to 1.
Step 2:	If $\lvert guess^2 - x \rvert < \varepsilon$, proceed to Step 4.
Step 3:	Set the value of guess to (x / guess + guess) / 2 and return to Step 2.
Step 4:	guess is the approximation of the square root.

It is necessary to test the *absolute* difference of `guess`^{2} and `x` against ε in Step 2, since the value of `guess` can approach the square root of `x` from either side.

Now that we have an algorithm for finding the square root at our disposal, it once again becomes a relatively straightforward task to develop a function to calculate the square root. For the value of ε in the following function, the value .00001 was arbitrarily chosen.

Program 8.8

```
/* Function to calculate the absolute value of a number */
#include <stdio.h>
float  absolute_value (float x)
{
     if ( x < 0 )
       x = -x;
     return (x);
}


/* Function to compute the square root of a number */

float  square_root (float x)
{
     float  epsilon = .00001;
     float  guess   = 1.0;

     while  ( absolute_value (guess * guess - x) >= epsilon )
       guess = ( x / guess + guess ) / 2.0;

     return (guess);
}


main ()
{
```

continues

Program 8.8 Continued

```
    printf ("square_root (2.0) = %f\n", square_root (2.0));
    printf ("square_root (144.0) = %f\n", square_root (144.0));
    printf ("square_root (17.5) = %f\n", square_root (17.5));
}
```

Program 8.8 OUTPUT

```
square_root (2.0) = 1.414216
square_root (144.0) = 12.000000
square_root (17.5) = 4.183300
```

(The actual values that are displayed by running this program on your computer system may differ slightly in the less significant digits.)

The above program requires a detailed analysis. The `absolute_value` function is defined first. This is the same function that was used in Program 8.7.

Next we find the `square_root` function. This function takes one argument called `x` and returns a value of type `float`. Inside the body of the function, two local variables called `epsilon` and `guess` are defined. The value of `epsilon`, which is used to determine when to end the iteration process, is set to .00001. The value of our `guess` at the square root of the number is initially set to 1.0. These initial values are assigned to these two variables each time that the function is called.

After the local variables have been declared, a `while` loop is set up to perform the iterative calculations. The statement that immediately follows the `while` condition will be repetitively executed as long as the absolute difference between $guess^2$ and `x` is greater than or equal to `epsilon`. The expression

```
guess * guess - x
```

is evaluated and the result of the evaluation is passed to the `absolute_value` function. The result returned by the `absolute_value` function is then compared against the value of `epsilon`. If the value is greater than or equal to `epsilon`, then the desired accuracy of the square root has not yet been obtained. In that case, another iteration of the loop is performed to calculate the next value of `guess`.

Eventually the value of `guess` will be close enough to the true value of the square root and the `while` loop will terminate. At that point, the value of `guess` will be returned to the calling program. Inside the `main` function, this returned value is passed to the `printf` function, where it is displayed.

You may have noticed that *both* the `absolute_value` function and the `square_root` function have formal parameters named `x`. The C compiler doesn't get confused, however, and keeps these two values distinct.

In fact, a function always has its own set of formal parameters. So the formal parameter `x` used inside the `absolute_value` function is distinct from the formal parameter `x` used inside the `square_root` function.

The same is true for local variables. We can declare local variables with the same name inside as many functions as we desire. The C compiler will not confuse the usage of these variables, since a local variable can only be accessed from within the function where it is defined. Another way of saying this is that the *scope* of a local variable is the function in which it is defined. (As you will see in Chapter 11, "Pointers," C does provide a mechanism for indirectly accessing a local variable from outside of a function.)

Based upon this discussion, you can understand that when the value of $guess^2$ `- x` is passed to the `absolute_value` function and assigned to the formal parameter `x`, this assignment has absolutely *no* effect on the value of `x` inside the `square_root` function.

Declaring Return Types and Argument Types

We mentioned earlier that the C compiler assumes that a function returns a value of type `int` as the default case. More specifically, whenever a call is made to a function, the compiler will assume that the function returns a value of type `int` unless either of the following has occurred:

1. The function has been defined in the program before the function call is encountered.
2. The value returned by the function has been *declared* before the function call is encountered.

In Program 8.8, the `absolute_value` function is defined before the compiler encounters a call to this function from within the `square_root` function. The compiler will know, therefore, that when this call is encountered, that the `absolute_value` function will return a value of type `float`. Had the `absolute_value` function been defined *after* the `square_root` function, then upon encountering the call to the `absolute_value` function the compiler would have assumed that this function returned an integer value. Most C compilers will catch this error and generate an appropriate diagnostic message.

In order to be able to define the `absolute_value` function *after* the `square_root` function (or even in another file—see Chapter 15), we must *declare* the type of the result returned by the `absolute_value` function *before* the function is called. The declaration can be made inside the `square_root` function itself, or outside

of any function. In the latter case, the declaration is usually made at the beginning of the program.

Not only is the function declaration used to declare the function's return type, but it is also used to tell the compiler how many arguments the function takes and what their types are.

To declare `absolute_value` as a function that returns a value of type `float` and that takes a single argument, also of type `float`, the following declaration would be used:

```
float  absolute_value (float);
```

As you can see, you just have to specify the argument type inside the parentheses, and not its name. You can optionally specify a "dummy" name after the type if you like:

```
float  absolute_value (float  x);
```

This name doesn't have to be the same as the one used in the function definition—the compiler ignores it anyway.

> **QUICK TIP**
>
> A foolproof way to write a function declaration is to simply use your text editor to make a copy of the first line from the actual definition of the function. Remember to place a semicolon at the end.

If the function doesn't take an argument, use the keyword `void` between the parentheses. If the function doesn't return a value, this fact can also be declared to thwart any attempts at using the function as if it does:

```
void  calculate_triangular_number (int  n);
```

If the function takes a variable number of arguments (such as is the case with `printf` and `scanf`), the compiler `must` be informed. The declaration

```
int printf (char *format, ...);
```

tells the compiler that `printf` takes a character *pointer* as its first argument (more on that later), and is followed by any number of additional arguments (the use of the `...`). `printf` and `scanf` are declared in the special file `stdio.h`. This is why you have been placing the following line at the start of each of your programs:

```
#include <stdio.h>
```


Without this line, the compiler can assume `printf` and `scanf` take a fixed number of arguments, which may result in incorrect code being generated.

The compiler will automatically convert your arguments to the appropriate types when a function is called, but only if you have placed the function's definition or have declared the function and its argument types before the call.

Here are some reminders and suggestions about functions:

1. Remember that, by default, the compiler assumes that a function returns an `int`.
2. When defining a function that returns an `int`, define it as such.
3. When defining a function that doesn't return a value, define it as `void`.
4. The compiler will convert your arguments to agree with the ones the function expects only if you have previously defined or declared the function.

> **QUICK TIP**
>
> To play it safe, declare all functions in your program, even if they are defined before they are called. (You may decide later to move them someplace else in your file or even to another file.)

Note that ANSI C also supports an alternate older format for defining and declaring functions. This format is described in Appendix A and will not be used in this text. Whenever we talk about defining or declaring a function, we are referring to the newer *prototyping* format as described in this chapter.

Checking Function Arguments

The square root of a negative number takes us away from the realm of real numbers and into the area of imaginary numbers. So what would happen if we were to pass a negative number to our `square_root` function? The fact is, the Newton-Raphson process would never converge; that is, the value of `guess` would not get closer to the correct value of the square root with each iteration of the loop. Therefore, the criteria set up for termination of the `while` loop would *never* be satisfied, and the program would go into an infinite loop. Execution of the program would have to be abnormally terminated by typing in some command or hitting a special key at the terminal (such as the *Delete* key under UNIX or *CTRL-C* under MS-DOS).

Obviously, modifying the program to correctly account for this situation is called for in this case. We could put the burden on the calling routine and mandate that it never pass a negative argument to the `square_root` function. While this approach might seem reasonable, it does have its drawbacks. Eventually, we would develop a program that used the `square_root` function but which forgot to check the argument before calling the function. If a negative number were then passed to the function, the program would go into an infinite loop as described and would have to be aborted. Not very nice!

A much wiser and safer solution to the problem is to place the onus of checking the value of the argument on the `square_root` function itself. In that way, the function would be "protected" from *any* program that used it. A reasonable approach to take would be to check the value of the argument `x` inside the function and then (optionally) display a message if the argument were negative. The function could then immediately return without performing its calculations. As an indication to the calling routine that the `square_root` function did not work as expected, a value not normally returned by the function could be returned.

The following is a modified `square_root` function, which tests the value of its argument and which also includes a prototype declaration for the `absolute_value` function as described in the previous section.

```
/* Function to compute the square root of a number.
   If a negative argument is passed, then a message
   is displayed and -1.0 is returned.                  */

float  square_root (float x)
{
    float  epsilon = .00001;
    float  guess   = 1.0;
    float  absolute_value (float  x);

    if ( x < 0 )
    {
        printf ("Negative argument to square_root.\n");
        return (-1.0);
    }

    while  ( absolute_value (guess * guess - x) >= epsilon )
           guess = ( x / guess + guess ) / 2.0;

    return (guess);
}
```

If a negative argument is passed to the above function, an appropriate message is displayed, and the value -1.0 is immediately returned to the calling routine. If the argument is not negative, then calculation of the square root proceeds as previously described.

As you can see from the modified `square_root` function, we can have more than one `return` statement in a function. Whenever a `return` is executed, control is immediately sent back to the calling function; any program statements in the function that appear after the `return` are not executed. This fact also makes the `return` statement ideal for use by a function that does not return a value. In such a case, the `return` statement takes the simpler form

```
return;
```

since no value is to be returned. Obviously, if the function *is* supposed to return a value, then this form cannot be used to return from the function.

Top-Down Programming

The notion of functions that call functions that in turn call functions, and so on, forms the basis for producing good structured programs. In the `main` routine of Program 8.8, the `square_root` function is called several times. All the details concerned with the actual calculation of the square root are contained within the `square_root` function itself, and not within `main`. Thus, we can write a call to this function before we even write the instructions of the function itself, as long as we specify the arguments that the function takes and the value that it returns.

Later, when proceeding to write the code for the `square_root` function, this same type of *top-down programming* technique can be applied: We can write a call to the `absolute_value` function without concerning ourselves at that time with the details of operation of that function. All we need to know is that we *can* develop a function to take the absolute value of a number.

The same programming technique that makes programs easier to write also makes them easier to read. Thus the reader of Program 8.8 can easily determine upon examination of the `main` routine that the program is simply calculating and displaying the square root of three numbers. He or she need not sift through all of the details of how the square root is actually calculated in order to glean this information. If the reader wishes to get more involved in details, then the specific code associated with the `square_root` function can be studied. Inside that function, the same discussion applies to the `absolute_value` function. The reader need not know how the absolute value of a number is calculated in order to understand the operation of the `square_root` function. Such details are relegated to the `absolute_value` function itself, which can be studied if a more detailed knowledge of its operation is desired.

Functions and Arrays

As with ordinary variables and values, it is also possible to pass the value of an array element and even an entire array as an argument to a function. To pass a single array element to a function (which is what we were doing in an array), the array element is specified as an argument to the function in the normal fashion. So, to take the square root of `averages[i]` and assign the result to a variable called `sq_root_result`, a statement such as

```
sq_root_result = square_root (averages[i]);
```

would do the trick.

Inside the `square_root` function itself, nothing special has to be done to handle single array elements passed as arguments. In the same manner as with a simple variable, the value of the array element will be copied into the value of the corresponding formal parameter when the function is called.

Passing an entire array to a function is an entirely new ball game. To pass an array to a function, it is only necessary to list the name of the array, *without any subscripts*, inside the call to the function. As an example, if we assume that `grade_scores` has been declared as an array containing 100 elements, then the expression

```
minimum (grade_scores)
```

will in effect pass the entire 100 elements contained in the array `grade_scores` to the function called `minimum`. Naturally, on the other side of the coin, the `minimum` function must be expecting an entire array to be passed as an argument and must make the appropriate formal parameter declaration. So the `minimum` function might look something like this:

```
int  minimum (int  values[100])
{
    ...
    return (minimum_value);
}
```

The declaration defines the function `minimum` as returning a value of type `int` and as taking as its argument an array containing 100 integer elements. References made to the formal parameter array `values` will reference the appropriate elements inside the array that was passed to the function. Based upon the function call previously shown and the corresponding function declaration, a reference made to `values[4]`, for example, would actually reference the value of `grade_scores[4]`.

Program 8.9

```
/* Function to find the minimum in an array */
#include <stdio.h>
int  minimum (int  values[10])
{
    int  minimum_value, i;

    minimum_value = values[0];

    for ( i = 1;  i < 10;  ++i )
        if ( values[i] < minimum_value )
           minimum_value = values[i];

    return (minimum_value);
}

main ()
{
    int  scores[10], i, minimum_score;
    int  minimum (int  values[10]);

    printf ("Enter 10 scores\n");

    for ( i = 0;  i < 10;  ++i )
        scanf ("%i", &scores[i]);

    minimum_score = minimum (scores);
    printf ("\nMinimum score is %i\n", minimum_score);
}
```

Program 8.9 OUTPUT

```
Enter 10 scores
69
97
65
87
69
86
78
67
92
90

Minimum score is 65
```

For our first program that illustrates a function that takes an array as an argument, let's write a function `minimum` to find the minimum value in an array of ten integers. This function, together with a `main` routine to set up the initial values in the array, is shown in Program 8.9.

The first thing that will catch your eye inside `main` is the prototype declaration for the `minimum` function. This tells the compiler that `minimum` returns an `int` and takes an array of ten `int`s. Remember, it's not really necessary to make this declaration here, since the `minimum` function is defined before it's called from inside `main`. However, we're going to play it safe throughout the rest of this text and declare all functions that are used.

After the array `scores` is defined, the user is prompted to enter ten values. The `scanf` call places each number as it is keyed in into `scores[i]`, where `i` ranges from 0 through 9. After all the values have been entered, the `minimum` function is called with the array `scores` as an argument.

The formal parameter name `values` is used to reference the elements of the array inside the function. It is declared to be an array of ten integer values. The local variable `minimum_value` is used to store the minimum value in the array and is initially set to `values[0]`, the first value in the array. The `for` loop sequences through the remaining elements of the array, comparing each element in turn against the value of `minimum_value`. If the value of `values[i]` is less than `minimum_value`, then a new minimum in the array has been found. In such a case, the value of `minimum_value` is reassigned to this new minimum value and the scan through the array continues.

When the `for` loop has completed execution, `minimum_value` is returned to the calling routine, where it is assigned to the variable `minimum_score` and is then displayed to the user.

With our general-purpose `minimum` function in hand, we can use it to find the minimum of *any* array containing ten integers. If we had five different arrays containing ten integers each, we could simply call the minimum function five separate times to find the minimum value of each array. And we can just as easily define other functions to perform tasks, such as finding the maximum value, the median value, the mean (average) value, and so on.

By defining small, independent functions that perform well-defined tasks, we can build upon these functions to accomplish more sophisticated tasks and also make use of them for other related programming applications. For example, we could define a function `statistics`, which takes an array as an argument and perhaps, in turn, calls a `mean` function, a `standard_deviation` function, and so on, to accumulate statistics about an array. This type of program methodology is the key to the development of programs that are easy to write, understand, modify, and maintain.

Of course, our general-purpose `minimum` function is not so general purpose in the sense that it only works on an array of precisely ten elements. But this problem is relatively easy to rectify. We can extend the versatility of this function by having it take the number of elements in the array as an argument. In the function declaration, we can then omit the specification of the number of elements contained in the formal parameter array. The C compiler actually ignores this part of the declaration anyway; all the compiler is concerned with is the fact that an array is expected as an argument to the function and not how many elements are in it.

Program 8.10 is a revised version of Program 8.9 in which the `minimum` function finds the minimum value in an integer array of arbitrary length.

Program 8.10

```
/* Function to find the minimum in an array */
#include <stdio.h>
int  minimum (int  values[], int  number_of_elements)
{
    int  minimum_value, i;

    minimum_value = values[0];

    for ( i = 1;  i < number_of_elements;  ++i )
        if ( values[i] < minimum_value )
            minimum_value = values[i];

    return (minimum_value);
}

main ()
{
    int  array1[5] = { 157, -28, -37, 26, 10 };
    int  array2[7] = { 12, 45, 1, 10, 5, 3, 22 };
    int  minimum (int  values[], int  number_of_elements);

    printf ("array1 minimum: %i\n", minimum (array1, 5));
    printf ("array2 minimum: %i\n", minimum (array2, 7));
}
```

Program 8.10 OUTPUT

```
array1 minimum: -37
array2 minimum: 1
```

This time the function `minimum` is defined to take two arguments: first, the array whose minimum we wish to find; and second, the number of elements in the array. The open and closed brackets that immediately follow `values` in the function header serve to inform the C compiler that `values` is an array of integers. As was stated, the compiler really doesn't need to know how large it is.

The formal parameter `number_of_elements` replaces the constant 10 as the upper limit inside the `for` statement. So the `for` statement sequences through the array from `values[1]` through the last element of the array, which is `values[number_of_elements - 1]`.

In the `main` routine, two arrays called `array1` and `array2` are defined to contain five and seven elements, respectively.

Inside the first `printf` call, a call is made to the `minimum` function with the arguments `array1` and 5. This second argument specifies the number of elements contained in `array1`. The `minimum` function finds the minimum value in the array and the returned result of -37 is then displayed at the terminal. The second time the minimum function is called, `array2` is passed, together with the number of elements in that array. The result of 1 as returned by the function is then passed to the `printf` function to be displayed.

Assignment Operators

Study the following program and try to guess the output *before* looking at the actual program results.

Program 8.11

```
#include <stdio.h>
void  multiply_by_two (float  array[], int  n)
{
    int  i;

    for ( i = 0;  i < n;  ++i )
        array[i] *= 2;
}

main ()
{

    float  float_values[4] = { 1.2, -3.7, 6.2, 8.55 };
    int    i;
    void   multiply_by_two (float  array[], int  n);

    multiply_by_two (float_values, 4);

    for ( i = 0;  i < 4;  ++i )
```



```
        printf ("%.2f   ", float_values[i]);

    printf ("\n");
}
```

Program 8.11 OUTPUT

```
2.40    -7.40    12.40    17.10
```

When you were examining Program 8.11, your attention surely must have been drawn to the statement

```
array[i] *= 2;
```

Hopefully, based upon the name of the function, you were able to determine what this statement is actually doing. This is certainly one of the primary reasons why meaningful function names should always be chosen. The effect of the so-called "times equals" operator (`*=`) is to multiply the expression on the left side of the operator by the expression on the right side of the operator and *to store the result back into the variable on the left side of the operator*. So the previous expression is equivalent to the statement

```
array[i] = array[i] * 2;
```

The "times equals" operator saves us from having to repeat what appears on the left side of the operator on the right side.

As you might have guessed, there are analagous operators for all binary arithmetic operators in C. These operators are called *assignment operators* and are formed by listing the normal binary operator immediately followed by the assignment operator =. So the expression

```
counter += 5
```

uses the "plus equals" assignment operator to add 5 to the value of `counter` and is equivalent to the expression

```
counter = counter + 5
```

A slightly more involved expression,

```
a /= b + c
```

divides a by whatever appears to the right of the equals sign—or by the sum of `b` and `c`—and stores the result back into `a`. The addition is performed first

because the addition operator has higher precedence than the assignment operator. In fact, all operators except the comma operator have higher precedence than the assignment operators, which all have the same precedence.

In this case, this expression is identical to

```
a = a / (b + c)
```

The motivation for using assignment operators is threefold. First, the program statement becomes easier to write, since what appears on the left side of the operator does not have to be repeated on the right side. Second, the resulting expression is usually easier to read. Third, the use of these operators can result in programs that execute faster on the computer.

As an example of an expression that is both easier to write and to read, consider the following program statement, which subtracts 10 from the value contained in `Board[row + col - 5]`:

```
Board[row + col - 5]  =  Board[row + col - 5] - 10;
```

With the use of the `-=` assignment operator, this statement could be written as

```
Board [row + col - 5] -= 10;
```

In the latter case, it is far easier to see that 10 is being subtracted from the indicated element in the `Board` array, since in the former case, a visual comparison must be made between the element on the left side of the equals sign and the element on the right side to determine that the same element of the `Board` array is being referenced.

Getting back to the main point to be made about the preceding program, you may have realized by now that the function `multiply_by_two` actually *changes* values inside the `float_values` array. Isn't this a contradiction to what we have stated before about a function not being able to change the value of its arguments? Not really.

This program example points out one major distinction that must always be kept in mind when dealing with array arguments.

> **QUICK TIP**
>
> If a function changes the value of an array element, then that change will be made to the original array that was passed to the function. This change remains in effect even after the function has completed execution and has returned to the calling routine.

The reason an array behaves differently from a simple variable or an array element—whose value *cannot* be changed by a function—is worthy of explanation. We stated that when a function is called, the values that are passed as arguments to the function are copied into the corresponding formal parameters. This statement is still valid. However, when dealing with arrays, the entire contents of the array are *not* copied into the formal parameter array. Instead, the function gets passed information describing *where* in the computer's memory the array is located. Any changes made to the formal parameter array by the function are actually made to the original array passed to the function, and not to a copy of the array. Therefore, when the function returns, these changes still remain in effect.

Remember, the preceding discussion applies only to entire arrays that are passed as arguments, and not to individual elements, whose values are copied into the corresponding formal parameters and therefore cannot be permanently changed by the function. Chapter 11, "Pointers," discusses these concepts in greater detail.

To further illustrate the idea that a function can change values in an array passed as an argument, we will develop a function to sort (rank) an array of integers. The process of sorting has always received much attention by computer scientists, probably because sorting is an operation that is so commonly performed. Many sophisticated algorithms have been developed in order to sort a set of information in the least amount of time, using as little of the computer's memory as possible. Since the purpose of this book is not to teach such sophisticated algorithms, we will develop a `sort` function that uses a fairly straightforward algorithm to sort an array into *ascending order*. Sorting an array into ascending order means rearranging the values in the array so that the elements progressively increase in value from the smallest to the largest. By the end of such a sort, the minimum value will be contained in the first location of the array, while the maximum value will be found in the last location of the array, with values that progressively increase in between.

If we wanted to sort an array of n elements into ascending order, we could do so by performing a successive comparison of each of the elements of the array. We could begin by comparing the first element in the array against the second. If the first element were greater in value than the second, then we could simply "swap" the two values in the array; that is, exchange the values contained in these two locations.

Next, we could compare the first element in the array (which we now know is less than the second) against the third element in the array. Once again, if the first value were greater than the third, we would exchange these two values. Otherwise, we would leave them alone. Now we would have the smallest of the first three elements contained in the first location of the array.

If we repeated the above process for the remaining elements in the array—comparing the first element against each successive element and exchanging their values if the former were larger than the latter—then the smallest value of the entire array would be contained in the first location of the array by the end of the process.

If we now did the same thing with the second element of the array, that is, compared it against the third element, then against the fourth, and so on; and if we exchanged any values that were out of order, we would then end up with the next smallest value contained in the second location of the array when the process was completed.

It should now be clear how we can go about sorting the array by performing these successive comparisons and exchanges as needed. The process will stop after we have compared the next-to-last element of the array against the last and have interchanged their values if required. At that point, the entire array will have been sorted into ascending order.

The following algorithm gives a more concise description of the above sorting process. We assume here that we are sorting an array *a* of *n* elements.

Simple Exchange Sort Algorithm

Step 1:	Set *i* to 0.
Step 2:	Set *j* to *i* + 1.
Step 3:	If *a[i]* > *a[j]*, exchange their values.
Step 4:	Set *j* to *j* + 1. If *j* < *n*, go to Step 3.
Step 5:	Set *i* to *i* + 1. If *i* < *n* - 1, go to Step 2.
Step 6:	*a* is now sorted in ascending order.

Program 8.12 implements the above algorithm in a function called `sort`, which takes two arguments: the array to be sorted and the number of elements in the array.

The `sort` function implements the algorithm as a set of nested `for` loops. The outermost loop sequences through the array from the first element to the next-to-last element (`a[n-2]`). For each such element, a second `for` loop is entered, which starts from the element after the one currently selected by the outer loop and ranges through the last element of the array.

If the elements are out of order (that is, if `a[i]` is greater than `a[j]`), then the elements are switched. The variable `temp` is used as a temporary storage place while the switch is being made.

When both `for` loops are finished, the array will have been sorted into ascending order. Execution of the function is then complete.

In the `main` routine, `array` is defined and initialized to 16 integer values. The program then displays the values of the array at the terminal and proceeds to call the `sort` function, passing as arguments `array` and 16, the number of elements in `array`. After the function returns, the program once again displays the values contained in `array`. As you can see from the output, the function successfully sorted the array into ascending order.

The `sort` function shown in Program 8.12 is fairly simple. The price that must be paid for such a simplistic approach is one of execution time. If we had to sort an extremely large array of values (arrays containing thousands of elements, perhaps), then the sort routine as we have implemented it here could take a considerable amount of execution time. If this happened, then we would have to resort to one of the more sophisticated algorithms that we alluded to in our discussions. *The Art of Computer Programming, Volume 3, Sorting and Searching* (Donald E. Knuth, Addison-Wesley) is probably the best reference source for such algorithms.

Program 8.12

```
/* Sort an array of integers into ascending order */
#include <stdio.h>
void  sort (int  a[], int  n)
{
    int  i, j, temp;

    for ( i = 0;  i < n - 1;  ++i )
        for ( j = i + 1;  j < n;  ++j )
            if ( a[i] > a[j] )
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
}

main ()
{
    int  i;
    int  array[16] = { 34, -5, 6, 0, 12, 100, 56, 22,
                       44, -3, -9, 12, 17, 22, 6, 11 };
    void sort (int  a[], int  n);

    printf ("The array before the sort:\n");
```

continues

Program 8.12 Continued

```
    for ( i = 0;  i < 16;  ++i )
        printf ("%i ", array[i]);

    sort (array, 16);

    printf ("\n\nThe array after the sort:\n");

    for ( i = 0;  i < 16;  ++i )
        printf ("%i ", array[i]);

    printf ("\n");
}
```

Program 8.12 OUTPUT

```
The array before the sort:
34 -5 6 0 12 100 56 22 44 -3 -9 12 17 22 6 11

The array after the sort:
-9 -5 -3 0 6 6 11 12 12 17 22 22 34 44 56 100
```

Multidimensional Arrays

A multidimensional array element may be passed to a function just as any ordinary variable or single-dimensional array element can. So the statement

```
square_root (matrix[i][j]);
```

will call the `square_root` function, passing the value contained in `matrix[i][j]` as the argument.

An entire multidimensional array can be passed to a function the same way that a single-dimensional array can: You simply list the name of the array. For example, if the matrix `measured_values` is declared to be a two-dimensional array of integers, the C statement

```
scalar_multiply (measured_values, constant);
```

could be used to invoke a function that multiplies each element in the matrix by the value of `constant`. This implies, of course, that the function itself may change the values contained inside the `measured_values` array. The discussion

of this topic for single-dimensional arrays also applies here: An assignment made to any element of the formal parameter array inside the function makes a permanent change to the array that was passed to the function.

When declaring a single-dimensional array as a formal parameter inside a function, it was stated that the actual dimension of the array was not needed; simply use a pair of empty brackets to inform the C compiler that the parameter is in fact an array. This does not totally apply in the case of multi-dimensional arrays. For a two-dimensional array, the number of rows in the array may be omitted, but the declaration *must* contain the number of columns in the array. So the declarations

```
int  array_values[100][50]
```

and

```
int  array_values[][50]
```

are both valid declarations for a formal parameter array called `array_values` containing 100 rows by 50 columns; but the declarations

```
int  array_values[100][]
```

and

```
int  array_values[][]
```

are not, since the number of columns in the array *must* be specified.

In the next program, we define a function `scalar_multiply`, which multiplies a two-dimensional integer array by a scalar integer value. We will assume for purposes of this example that the array is dimensioned 3 x 5. The `main` routine will call the `scalar_multiply` routine twice. After each call, the array will be passed to the `display_matrix` routine to display the contents of the array. Pay careful attention to the nested `for` loops that are used in both `scalar_mutiply` and `display_matrix` to sequence through each element of the two-dimensional array.

Program 8.13

```
#include <stdio.h>
main ()
{

    void  scalar_multiply (int  matrix[3][5], int  scalar);
    void  display_matrix (int  matrix[3][5]);
```

continues

Program 8.13 Continued

```
        {
            {  7, 16, 55, 13, 12 },
            { 12, 10, 52,  0,  7 },
            { -2,  1,  2,  4,  9 }
        };

    printf ("Original matrix:\n");
    display_matrix (sample_matrix);

    scalar_multiply (sample_matrix, 2);

    printf ("\nMultiplied by 2:\n");
    display_matrix (sample_matrix);

    scalar_multiply (sample_matrix, -1);

    printf ("\nThen multiplied by -1:\n");
    display_matrix (sample_matrix);
}

/* Function to multiply a 3 x 5 array by a scalar */

void  scalar_multiply (int  matrix[3][5], int  scalar)
{
   int  row, column;

   for ( row = 0;  row < 3;  ++row )
      for ( column = 0;  column < 5;  ++column )
         matrix[row][column]  *=  scalar;
}


void  display_matrix (int  matrix[3][5])
{
   int   row, column;

   for ( row = 0;  row < 3;  ++row)
   {

      for ( column = 0;  column < 5;  ++column )
         printf ("%5i", matrix[row][column]);

      printf ("\n");
   }
}
```


Program 8.13 OUTPUT

```
Original matrix:
    7   16   55   13   12
   12   10   52    0    7
   -2    1    2    4    9

Multiplied by 2:
   14   32  110   26   24
   24   20  104    0   14
   -4    2    4    8   18

Then multiplied by -1:
  -14  -32 -110  -26  -24
  -24  -20 -104    0  -14
    4   -2   -4   -8  -18
```

The `main` routine defines the matrix `sample_values` and then calls the `display_matrix` function to display its initial values at the terminal. Inside the `display_matrix` routine, you will notice the nested `for` statements. The first or outermost `for` statement sequences through each row in the matrix, so the value of the variable `row` varies from 0 through 2. For each value of `row`, the innermost `for` statement is executed. This `for` statement sequences through each column of the particular row, so the value of the variable `column` ranges from 0 through 4.

The `printf` statement displays the value contained in the specified row and column using the format characters `%5i` to ensure that the elements line up in the display. After the innermost `for` loop has finished execution—meaning that an entire row of the matrix has been displayed—a newline character is displayed so that the next row of the matrix is displayed on the next line of the terminal.

The first call to the `scalar_mutiply` function specifies that the `sample_matrix` array is to be multiplied by 2. Inside the function, a simple set of nested `for` loops is set up to sequence through each element in the array. The element contained in `matrix[row][column]` is multiplied by the value of `scalar` in accordance with the use of the assignment operator `*=`. After the function returns to the `main` routine, the `display_matrix` function is once again called to display the contents of the `sample_matrix` array. The program's output verifies that each element in the array has in fact been multiplied by 2.

The `scalar_multiply` function is called a second time to multiply the now modified elements of the `sample_matrix` array by -1. The modified array is then displayed by a final call to the `display_matrix` function, and program execution is then complete.

Global Variables

We are now ready to tie together many of the principles that we have learned in this chapter, as well as learn some new ones. What we would like to do now is take Program 7.7, which converted a positive integer to another base, and rewrite it in function form. In order to do this, we must conceptually divide the program into logical segments. If you glance back at that program you will see that this is readily accomplished simply by looking at the three comment statements inside main. They suggest the three primary functions that the program is performing: getting the number and base from the user, converting the number to the desired base, and then displaying the results.

We can define three functions to perform an analogous task. The first function we will call get_number_and_base. This function will prompt the user to enter the number to be converted and the base and will read these values in from the terminal. Here we will make a slight improvement over what was done in Program 7.7. If the user types in a value of base that is less than 2 or greater than 16, then the program will display an appropriate message at the terminal and set the value of the base to 10. In this manner, the program will end up redisplaying the original number to the user. (Another approach might be to let the user reenter a new value for the base, but this is left as an exercise.)

The second function we will call convert_number. This function will take the value as typed in by the user and convert it to the desired base, storing the digits resulting from the conversion process inside the converted_number array.

The third and final function we will call display_converted_number. This function will take the digits contained inside the converted_number array and display them to the user in the correct order. For each digit to be displayed, a lookup will be made inside the base_digits array so that the correct character is displayed for the corresponding digit.

The three functions that we will define will communicate with each other by means of *global variables*. It has been previously noted that one of the fundamental properties of a local variable is that its value can be accessed only by the function in which the variable is defined. As you might expect, this restriction does not apply to global variables. That is, a global variable's value can be accessed by any function in the program.

The distinguishing quality of a global variable declaration versus a local variable declaration is that the former is made outside of any function. Thus indicates the global nature—it does not belong to any particular function. Any function in the program can then access the value of that variable and can also change its value if desired.

In Program 8.14, four global variables are defined. Each of these variables is used by at least two functions in the program. Since the base_digits array and the variable next_digit are used exclusively by the function display_converted_number, they are *not* defined as global variables. Instead, these variables are locally defined within the function display_converted_number.

The global variables are defined first in the program. Since they are not defined within any particular function, the C system classifies these variables as global variables, which means, as we have mentioned, that they can now be referenced by any function in the program.

Program 8.14

```
/* Program to convert a positive integer to another base */
#include <stdio.h>
int       converted_number[64];
long int  number_to_convert;
int       base;
int       index = 0;

void  get_number_and_base (void)
{
    printf ("Number to be converted? ");
    scanf ("%li", &number_to_convert);

    printf ("Base? ");
    scanf ("%i", &base);

    if  ( base < 2  ||  base > 16 )
    {
        printf ("Bad base - must be between 2 and 16\n");
        base = 10;
    }
}
void  convert_number (void)
{
    do
    {
        converted_number[index] = number_to_convert % base;
        ++index;
        number_to_convert /= base;
    }
    while  ( number_to_convert != 0 );
}

void  display_converted_number (void)
{
```

continues