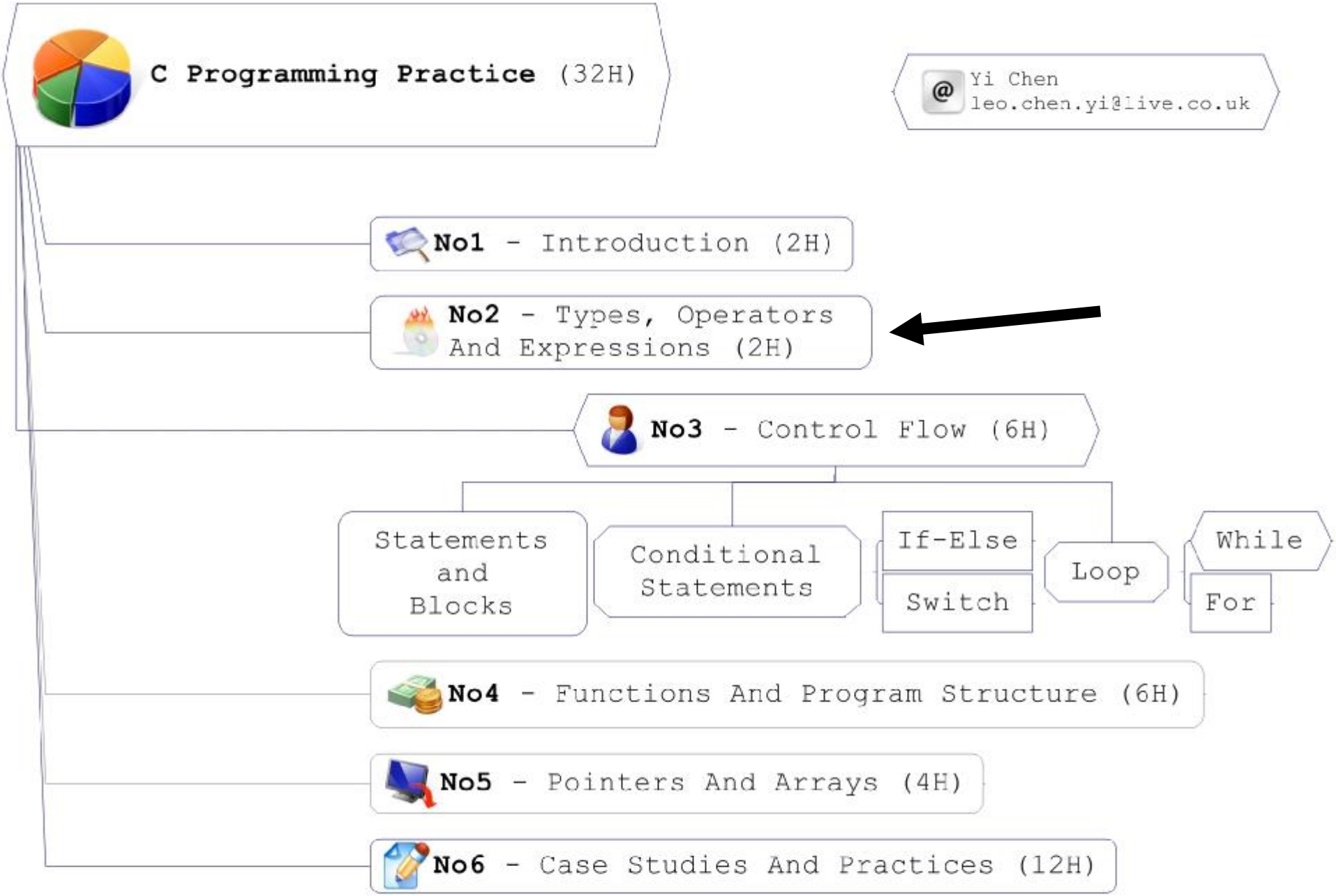
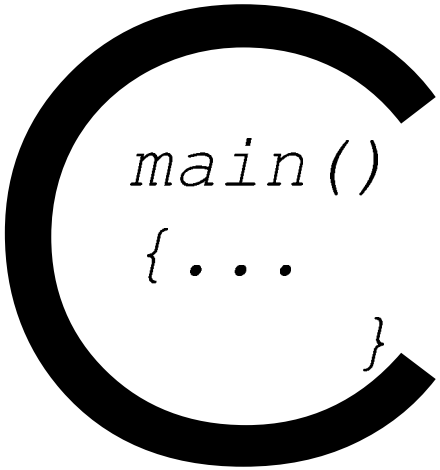

C Programming Practice (32H)
Yi Chen
leo.chen.yi@live.co.uk
No1 - Introduction (2H)
No2 - Types, Operators And Expressions (2H)
No3 - Control Flow (6H)
Statements and Blocks
Conditional Statements
If-Else
Switch
Loop
While
For
No4 - Functions And Program Structure (6H)
No5 - Pointers And Arrays (4H)
No6 - Case Studies And Practices (12H)

C Programming Practice
No[2]

Chen , Yi
leo.chen.yi@live.co.uk
30-Jul-2010



Types, Operators and Expressions

BETA 1.0.0.1

Contents

Contents

Variable Names

```
float    distance = 10.0;
#define  PI 3.14
```

Traditional C practice is to use **lower** case for variable names, and all **upper** case for symbolic constants.

1) Names are made up of: 1) **Letters**, a ~ z, A~Z ; 2) **Digits**, 0 ~ 9 ; 3) **Underscore**, _

2) The **first** character **must** be a letter, the underscore ``_'' counts as a letter

3) Keywords (e.g., for,while etc.) **cannot** be used as variable names

4) Variable names are **case sensitive**, e.g. int x; int X declares two different variables.

Variable Names

Examples of invalid C++ identifiers:

1AB3	(begins with a number)
E*6	(contains a special character)
while	(this is a keyword)

Good naming(meaningful)

force, mass, acceleration
population, density, heat

Bad naming (meaningless)

a, aa, aaa, Aa, b, c
a1, a2, a3, i,j,k

Exercise-4

Tell it correct or not:

```
int  money$owed;
int  total_count
int  score2
int  2ndscore
int  long

float _is_flag_
```

Static variables

static keyword has two meanings, depending on where the static variable is declared

Outside a function, static variables/functions **only** visible **within** that file, not globally

Inside a function, **static** variables:

- are still **local** to that function
- are initialised **only** during program initialisation
- do not get re-initialised with each function call

e.g.

```
static int somevar  =  0;
```

Register variables

During **execution**, data processed in **registers**

Explicitly store commonly used data in registers – **minimise** load/store overhead

Can explicitly declare certain variables as registers using register keyword:

- **must** be a simple type (implementation-dependent)
- **only** local variables and function arguments eligible
- excess/unallowed register declarations **ignored**, compiled as regular variables

Registers do **not** reside in addressed memory; **pointer** of a register variable illegal

Variable scope

> **scope** – the region in which a variable is valid

> Many cases, corresponds to **block** with variable's declaration

> Variables declared **outside** of a function have **global** scope
Function definitions also have scope

> Controlling program **flow** using conditional statements and loops

> **Dividing** a **complex** program into many **simpler** sub-programs using functions and modular programming techniques

Contents

Data Types and Sizes

C has a small family of datatypes:

> **Numeric**
  int, an integer, typically reflecting the natural size of integers on the host machine

  float, single-precision floating point

  Double, double-precision floating point

> **Character**
  char, a single byte, capable of holding one character in the local character set

> **User defined** (struct, union)

Data Types and Sizes

char: a single byte, capable of holding one character in the local character set

int: an integer, typically reflecting the natural size of integers on the host machine

float: single-precision floating point

double: double-precision floating point

	signed	**unsigned**
short	**short int** *length*; **short** *length*	**unsigned short** *length*; **unsigned short int** *length*
default	**int** *length*	**unsigned int** *length*
long	**long** *length*	**unsigned long** *length*
float	**float** *length*	N/A
double	**double** *length*	N/A
char	**char** *msg*; **signed char** *msg*	**unsigned char** *msg*

Compound data types

>> ***struct*** - structure containing one or multiple fields, each with its own type (or compound type)

- size is combined size of all the fields, padded for byte alignment
- anonymous or named

>> ***union*** - structure containing one of several fields, each with its own type (or compound type)

- size is size of **largest** field
- anonymous or named

Bit fields - structure fields with width in bits

- aligned and ordered in architecture-dependent manner
- can result in inefficient code

Compound data types

Define structure data type: *plot_xydata_s*

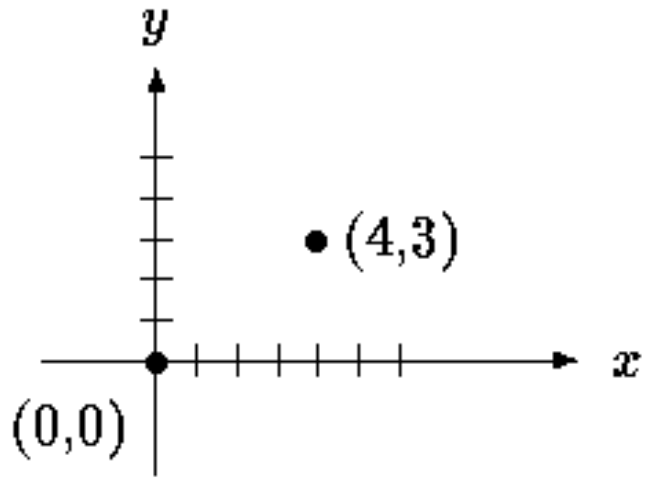
Generally,

in a ***header file .h***

```
typedef struct xydata_s
{
        double          *xdata ;
        double          *ydata ;

        char             xdata_unit[MAX_STRING_LEN+1] ;
        char             ydata_unit[MAX_STRING_LEN+1] ;

        int              data_length ;

        double           x_axis_length ;
        double           y_axis_length ;
        double           x_margin ;

        joint_driver_t driver_type;
} plot_xydata_t, *plot_xydata_p_t;
```

Define structure *name* and a *pointer*

Compound data types



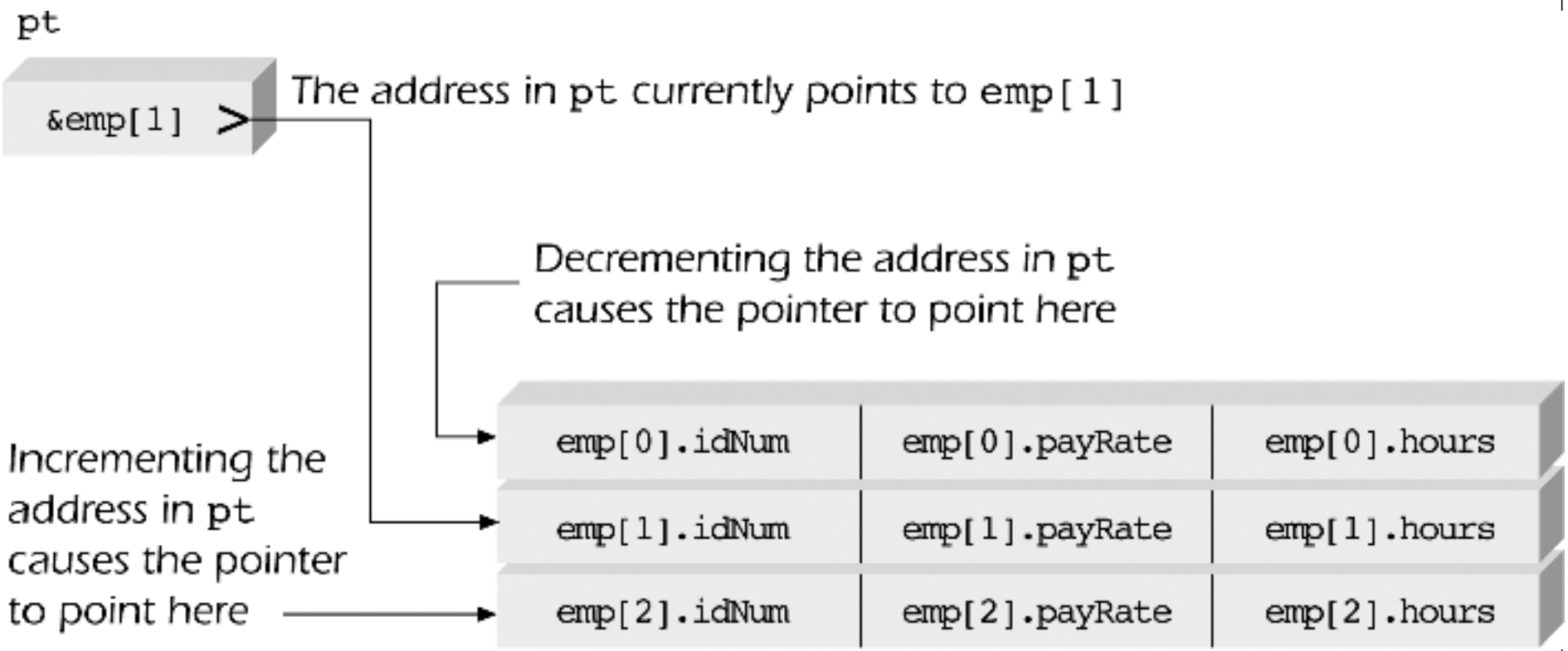

```
/* Not Correct */
typedef structure
{
int creation_status;

double origin[3];

}  XXX_data_t, *XXX_data_p_t;
```

```
/* Correct */
typedef structure XXX_data_s
{
  int creation_status;

  double origin[3];
}  XXX_data_t, *XXX_data_p_t;
```

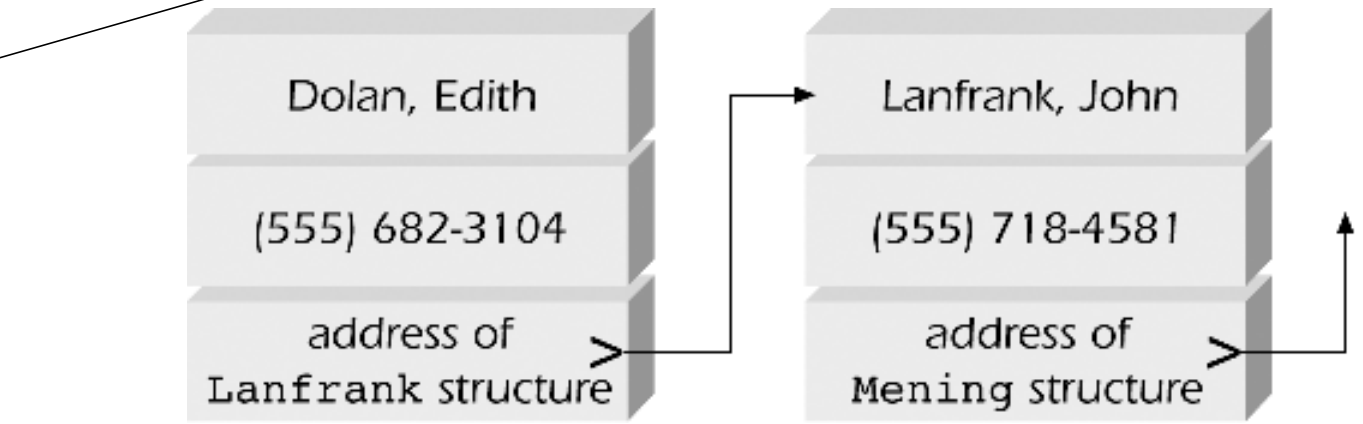
Compound data types

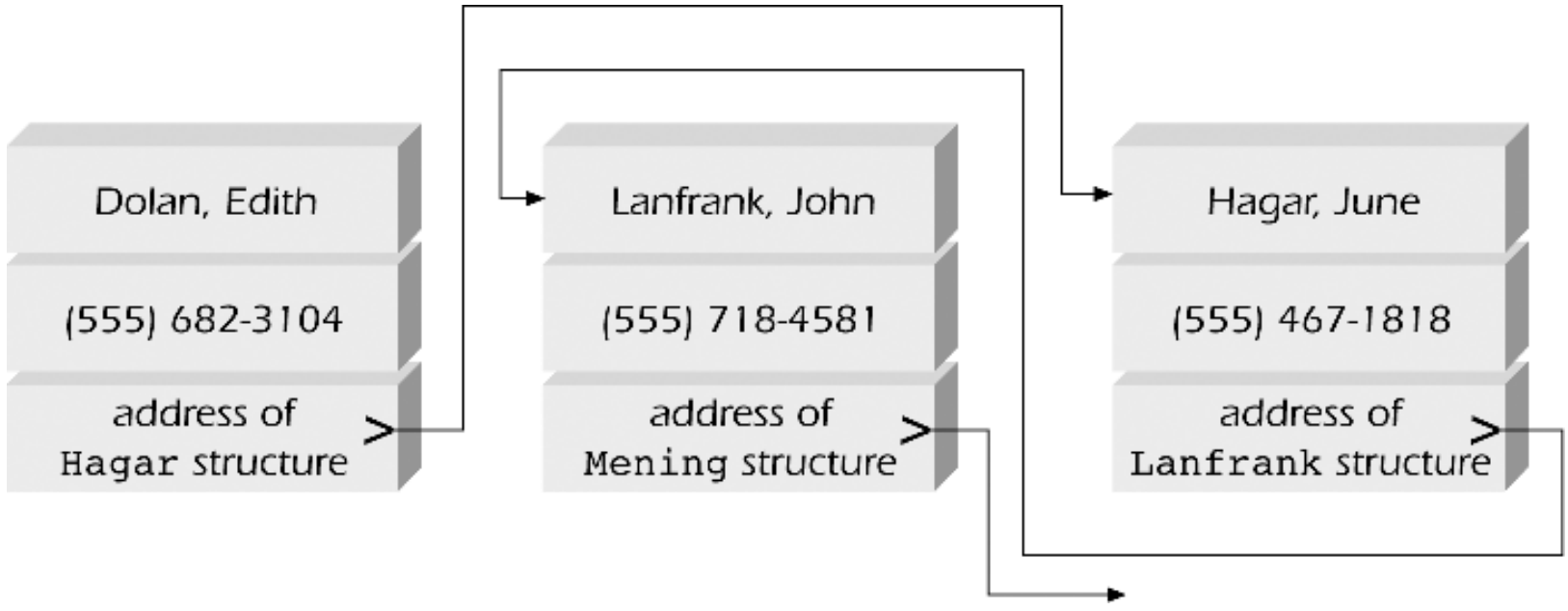
```
enum  TYPE {
                        SQUARE,
                        RECT,
                        CIRCILE,
                        POLYGON    } ;

struct  shape {

          float   params[MAX];

          enum  TYPE  CADtype ;
} ;
```

Compound data types

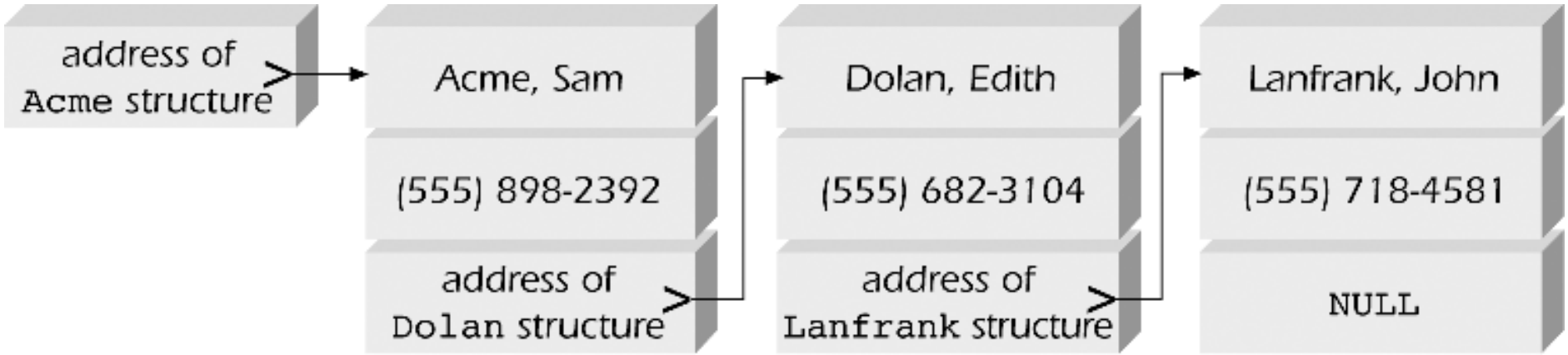


Linked Lists





Linked Lists



Contents

Constants

Constants are **literal/fixed** values assigned to variables or used directly in expressions.

```
#define PI      3.14
#define ERROR   1
#define OK      0
#define STREOF  '\0'

#define MAXLINE 1000
#define STEP    1

#define MSG     "Hello, world"

#define VTAB    '\013' /* ASCII vertical tab */
#define BELL    '\007' /* ASCII bell character */

or, in hexadecimal,

#define VTAB    '\xb' /* ASCII vertical tab */
#define BELL    '\x7' /* ASCII bell character */
```

Constants

The complete set of escape sequences is

\a	*alert (bell) character*	****	*backslash*
\b	*backspace*	**\?**	*question mark*
\f	*formfeed*	**\'**	*single quote*
\n	*newline*	**\"**	*double quote*
\r	*carriage return*	***ooo***	*octal number*
\t	*horizontal tab*	**\x*hh***	*hexadecimal number*
\v	*vertical tab*		

Constants

Integer constants may be written in **decimal**:

130, 45, 88203

An integer constant that begins with **0** (**zero**) is an **octal** number:

0130, 045

A sequence of digits preceded by **0X** or **0x** is a **hexadecimal** number:

0x90A, 0Xf2

(Either lowercase **a** through **f** or uppercase **A** through **F** is **acceptable**)

Constants

A **floating-point** constant consists of a string of digits (**integral** part) followed by a decimal point followed by a string of digits (**fractional** part) followed by an integer exponent.

The integer exponent is **e** or **E** optionally followed by **+** or followed by a string of **digits**.

Examples of floating-point constants are:

2.0, 4.3e4, 9.21E9, 13.E+4, 4e3, .390

Constants

A floating-point constant may be terminated by **f** or F to indicate that it is a float or by **l** or **L** to indicate that it is a **long double**.

If a floating-point constant is **not** terminated with either **f**, **F**, **l**, or **L**, it is of type **double**.

A **char constant** is delimited by single quotation marks, for example, '**Y**'.

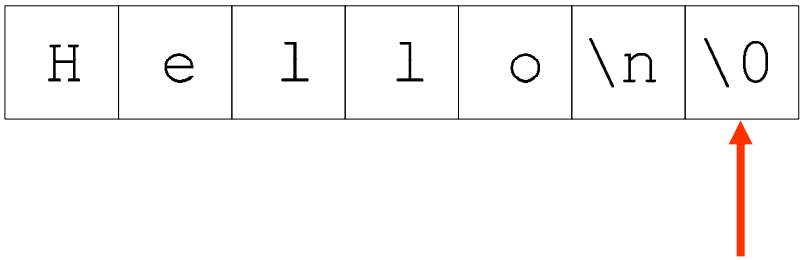
A **string constant** is delimited by **double quotation** marks, for example,

"This is a string constant."

Constants

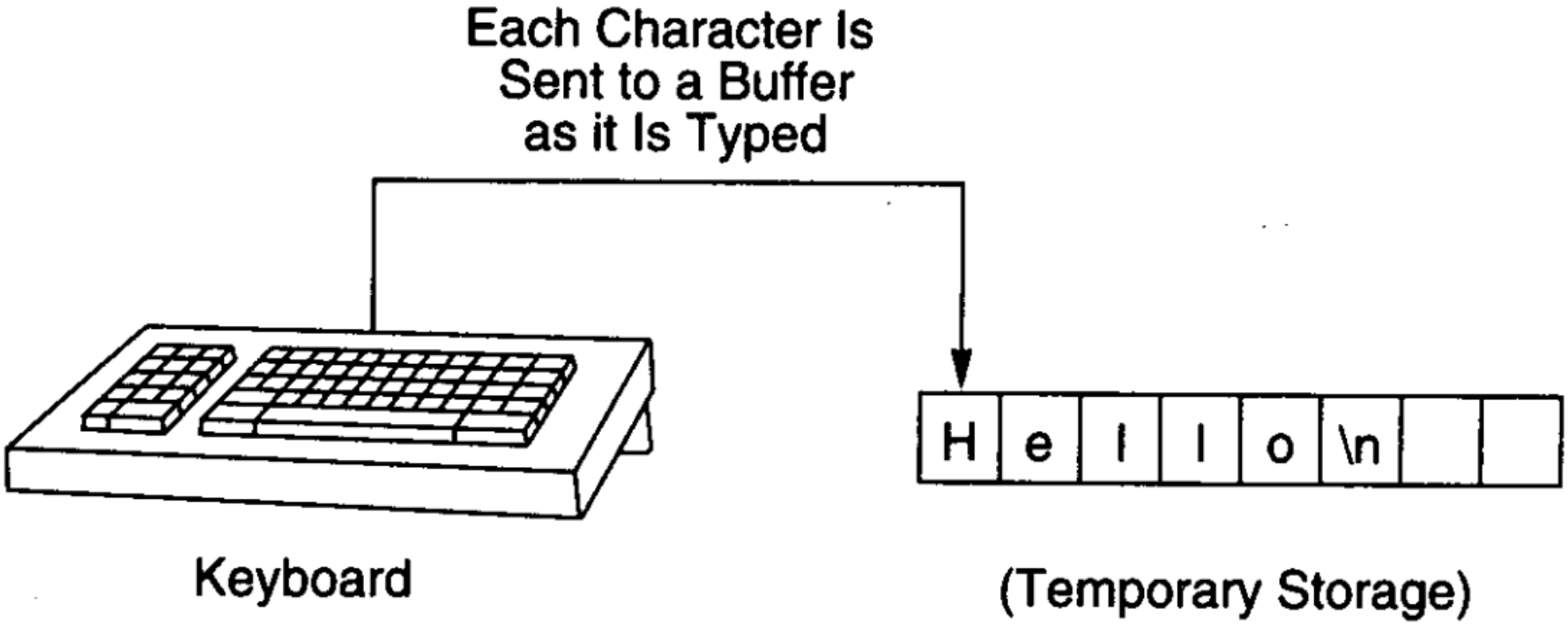
The internal representation of
a string has a null character '**\0**' at the **end**

"Hello\n"




```
  #define STREND '\0'

/* calculate string length */
  int strlen( char strin[] /*I*/)
  {
      int ind = 0;

      while ( strin[ind] != STREND )
          ++ind;

      return ind;
  }
```

Constants


Each Character Is
Sent to a Buffer
as it Is Typed
H
e
l
l
o
\n
Keyboard
(Temporary Storage)

Constants

Enumerations provide a convenient way to associate constant values with names, an alternative to **#define**.

There is one other kind of constant, the enumeration constant. An enumeration is a list of constant integer values, as in:

```
enum boolean { NO, YES };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };

          /* FEB = 2, MAR = 3, etc. */
```

Names in different enumerations **must** be distinct.

Letter	ASCII Code	EBCDIC Code	Letter	ASCII Code	EBCDIC Code
A	01000001	11000001	N	01001110	11010101
B	01000010	11000010	O	01001111	11010110
C	01000011	11000011	P	01010000	11010111
D	01000100	11000100	Q	01010001	11011000
E	01000101	11000101	R	01010010	11011001
F	01000110	11000110	S	01010011	11100010
G	01000111	11000111	T	01010100	11100011
H	01001000	11001000	U	01010101	11100100
I	01001001	11001001	V	01010110	11100101
J	01001010	11010001	W	01010111	11100110
K	01001011	11010010	X	01011000	11100111
L	01001100	11010011	Y	01011001	11101000
M	01001101	11010100	Z	01011010	11101001

Exercise-5

```
> int  x = 017; int y = 12; /* is  x > y? */

> short int  s=0xFFFF12; /* correct? */

> char c=-1;unsigned char uc=-1; /* correct? */

> puts("hel"+"lo"); puts("hel""lo");/*which is correct?*/

> enum userlist{S=0,L=3,XL}; /*what is the value of XL?*/

> enum userlist {S=0,L=-3,XL}; /*what is the value of XL?*/
```

Contents

Declarations

The general format for a declaration is:

type variable-name = value

```
char msg;              /*  uninitialised */
char msg = '1';        /* intialised  to  'A'*/
char msg1= 'A',msg2 = 'B';
                       /* multiple variables  initialised */
char msg1 = msg2 = 'Z';
                       /* multiple  initialisations  */

char  esc = '\\';

int   index  = 0;
int   limit  = MAXLINE+1;

float eps         = 1.0e-5;
const double e    = 2.71828182845905;
const char msg[] = "warning: ";

int strlen( const char[] );
```

Contents

Arithmetic Operators

operator	meaning	examples
+	addition	Workdays = 3 + 2 /* constant */ day1 + day2 + 7 /* variables, constants*/
-	substraction	3 – 2 /* constant */ int rest = workdays – 3 – 2 /* both */
*	multiplication	double work = force*distance;
/	division	double radius_sq = area/pi
%	modulus	int day = 7; int daymod = day%10

Assignment Operations

- **Increment operator ++:** unary operator for the special case when a variable is increased by 1
- **Prefix increment operator** appears before the variable

  Example: **++index**
- **Postfix increment operator** appears after the variable

  Example: **index ++**

Assignment Operations

- **Decrement operator --:** unary operator for the special case when a variable is decreased by 1

- **Prefix decrement operator** appears before the variable

  Example: **--index;**

- **Postfix decrement operator** appears after the variable

  Example: **index--;**

Assignment Operations

```
Example:    kdx = ++ndx;       //prefix increment

  is equivalent to
      ndx = ndx + 1;         //increment ndx first
      kdx = ndx;         //assign ndx's value to kdx

Example:    kdx = ndx++;       //postfix increment

 is equivalent to
      kdx = ndx;         //assign ndx's value to kdx
      ndx = ndx + 1;           //and then increment ndx
```

Contents

Relational Operators

Relational operators **compare** two operands to produce a '**boolean**' result.

In C any **non-zero** value (1 by convention) is considered to be '**true**' and **0** is considered to be **false**.

operator	meaning	examples
>	Greater than	3 > 2 /* 1 */ 2.99 > 3 /* 0 */
>=	Greater than or equal to	3 >= 2 /* 1 */ 2.99 >= 3 /* 0 */
<	Smaller than	3 < 2 /* 0 */ 2.99 < 3 /* 1 */
<=	Smaller than or equal to	3 <= 3 /* 1 */ 3.99 <= 3 /* 0 */

Relational Operators

operator	meaning	examples
==	equal to	3 == 2 /* 0 */ 3 == 3 /* 1 */ 'A' == 'A' /* 1 */ 'a' == 'A' /* 0 */
!=	not equal to	3 != 2 /* 1 */ 3 != 3 /* 0 */

Note:

(1) the "==" equality operator is different from the "=", assignment operator.

(2) the "==" operator on float variables is tricky because of finite precision.

(3) The unary negation operator ! converts a non-zero operand into 0, and a zero operand in 1.

Logical operators

The evaluation of an expression is **discontinued,** if the value of a conditional expression can be **determined early**. Be careful of any side effects in the code.

(3==3)||((c=getchar())==’y’). The **second** expression is **not** evaluated.

(0) && ((x=x+1)>0) The **second** expression is **not** evaluated.

operator	**meaning**	**examples**
&&	AND	(3 == 2) && (3 == 3) /* 0 */ (‘A’ == ‘A’) && (‘a’ == ‘a’) /* 1 */
||	OR	(3 == 2) || (3 == 3) /* 1 */ (‘A’ == ‘A’) || (‘a’ == ‘a’) /* 1 */
!	NOT	!(3 == 3) /* 0 */ !(2.99 >= 3) /* 1 */

Contents

Increment and decrement operators

```
for (idx=1; idx<=max_column; idx++)
  {
   for (jdx=1; jdx<=max_row; jdx++)
     {
       matrix[idx,jdx] = ...
                                    }
   ...
                                        }
```

> **idx++** is a short cut for idx = idx + 1

> **idx--** is a short cut for idx = idx – 1

> **jdx = idx++** is a short cut for jdx = idx ; idx = idx + 1.

idx is evaluated before it is **incremented**.

> **jdx = idx--** is a short cut for jdx = idx ; idx = idx - 1.

idx is evaluated before it is **decremented**.

Increment and decrement operators

cp_case_power.c

```
/* delete_keyword_from_string: delete all c from s */

  int delete_keyword_from_string(
  char str[],  /*I*/
  char keyword /*I*/
  )
  {
     int idx = 0;
     int jdx = 0;

     for (idx = jdx = 0; str[idx] != '\0'; idx++)
         if (str[idx] != keyword)
             str[jdx++] = str[idx];
     str[jdx] = '\0';

    return jdx;
  }
```

Contents

Bitwise Operators

AND is true only if both operands are true.
OR is true if any operand is true.
XOR is true if only one of the operand is true.

operator	meaning	examples
&	AND	0x77 & 0x3; /* evaluates to 0x3 */ 0x77 & 0x0; /* evaluates to 0 */
|	OR	0x700 | 0x33; /* evaluates to 0x733 */ 0x070 | 0 /* evaluates to 0x070 */
^	XOR	0x770 ^ 0x773; /* evaluates to 0x3 */ 0x33 ^ 0x33; /* evaluates to 0 */
>>	Right shift	0x010>>4; /* evaluates to 0x01 */ 4 >> 1 /* evaluates to 2 */
<<	Left shift	0x01<<4; /* evaluates to 0x10 */ 1<<2; /* evaluates to 4 */

Contents

Assignment Operators

Another common expression type found while programming in C is of the **type var = var** *operator* **expression**:

```
float  pi           = 3.14
float  displacement = time * velocity;
double area         = PI*radius*radius;
```

C provides **compact** assignment operators that can be used Instead:

```
 day+= 1  /*is the same as day = day+1  */
 day-=1   /*is the same as day = day-1  */
 day*=10  /*is the same as day = day*10 */
 day/=2   /*is the same as day = day/2  */
 day%=2   /*is the same as day = day%2  */
```

Contents

Type Conversions

When variables are promoted to **higher precision**, data is preserved. This is automatically done by the compiler for mixed data type expressions.

```
int     day     = 7   ;
float   workday = 0.0 ;

workday = day + 3.14159;
/ * day is promoted to float,
    workday = day (float)+ 3.14159 */
```

Type Conversions

Another conversion by the compiler is ’**char**’ **to** ’**int**’.

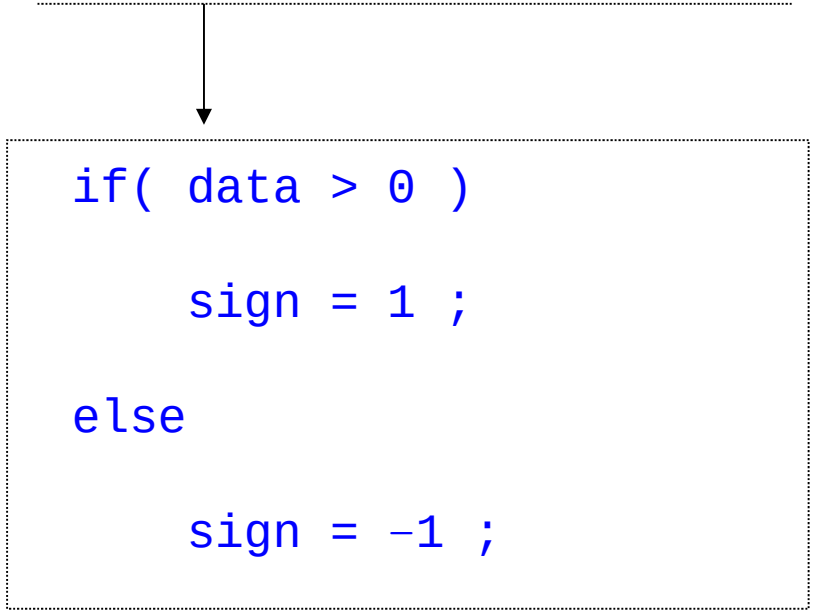
A char is just **a small integer**, so chars may be freely used in arithmetic expressions. This permits considerable flexibility in certain kinds of character transformations. One is exemplified by this naive implementation of the function **atoi**, which converts a string of digits into its numeric equivalent.

```
 /* atoi:  convert string to integer */
   int atoi( char str[] /*I*/)
   {
      int idx     = 0;
      int int_out = 0;

      for (idx = 0; str[idx] >= '0' && str[idx] <= '9'; ++idx)
           int_out = 10 * int_out + (str[idx] - '0');

       return int_out;                                    }
```

Type Conversions

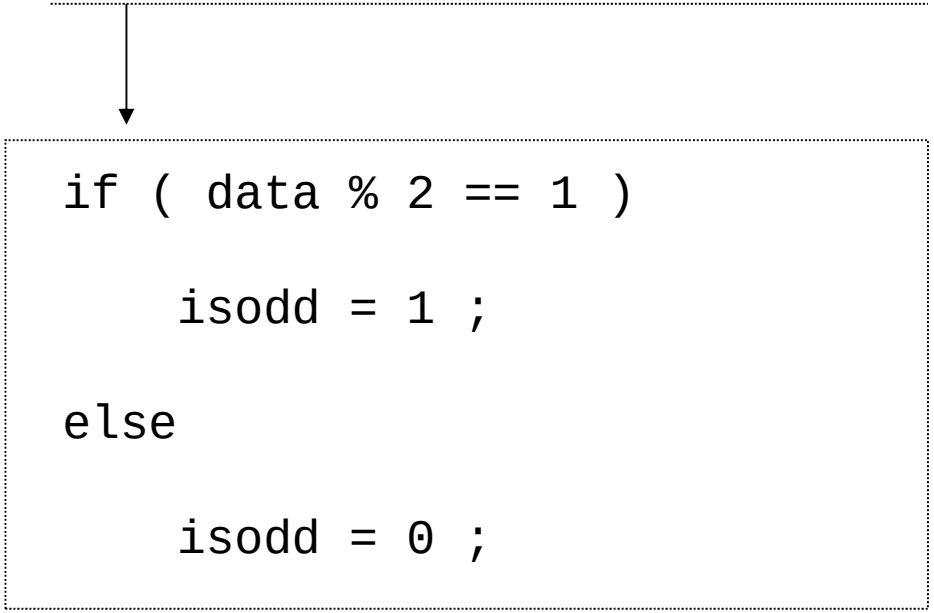
C provides syntactic sugar to express the same using the ternary operator '**? :**'

```
sign = data > 0 ? 1 : -1;
```



```
if( data > 0 )
    sign = 1 ;
else
    sign = -1 ;
```

```
isodd = data % 2 == 1 ? 1 : 0;
```



```
if ( data % 2 == 1 )
    isodd = 1 ;
else
    isodd = 0 ;
```

Contents

Order of Evaluation

++,–,(),sizeof have the highest priority

***,/,% have higher priority than +,**

==,!= have higher priority than &&,||

assignment operators = have very low priority

Note: Use()generously to avoid ambiguities
or side effects associated with order of operators.

```
displacement = velocity*time + 2
/* same as displacement = (velocity*time) + 2 */

flatx != 0 && flaty == 0
/* same as ( flatx != 0 ) && (flaty == 0 ) */

daya = dayb >= '0' && dayb <= '9'
/* same as daya = (dayb >= '0' ) && ( dayb <= '9' ) */
```

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (*type*) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

C Operators

> **arithmetic** (+, -, *, /, %)
> **assignment** (=) and augmented assignment (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=)
> **bitwise logic** (~, &, |, ^)
> **bitwise shifts** (<<, >>)
> **boolean logic** (!, &&, ||)
> **conditional evaluation** (? :)
> **equality testing** (==, !=)
> **function argument collection** (( ))
> **increment and decrement** (++, --)
> **member selection** (., ->)
> **object size** (sizeof)
> **order relations** (<, <=, >, >=)
> **reference and dereference** (&, *, [ ])
> **sequencing** (,)
> **subexpression grouping** (( ))
> **type conversion** ((typename))

Exercise-6

Determine the value of the following floating point expressions:

```
a.  3.0 + 4.0 * 6.0
b.  3.0 * 4.0 /  6.0 + 6.0
c.  2.0 * 3.0 /  12.0 * 8.0 /  4.0
d.  10.0 * (1.0 + 7.0 * 3.0)
e.  20.0 -  2.0 /  6.0 + 3.0
f.  20.0 -  2.0 /  (6.0 + 3.0)
g.  (20.0 -  2.0) /  6.0 + 3.0
h.  (20.0 -  2.0) /  (6.0 + 3.0)
```

Exercise-7

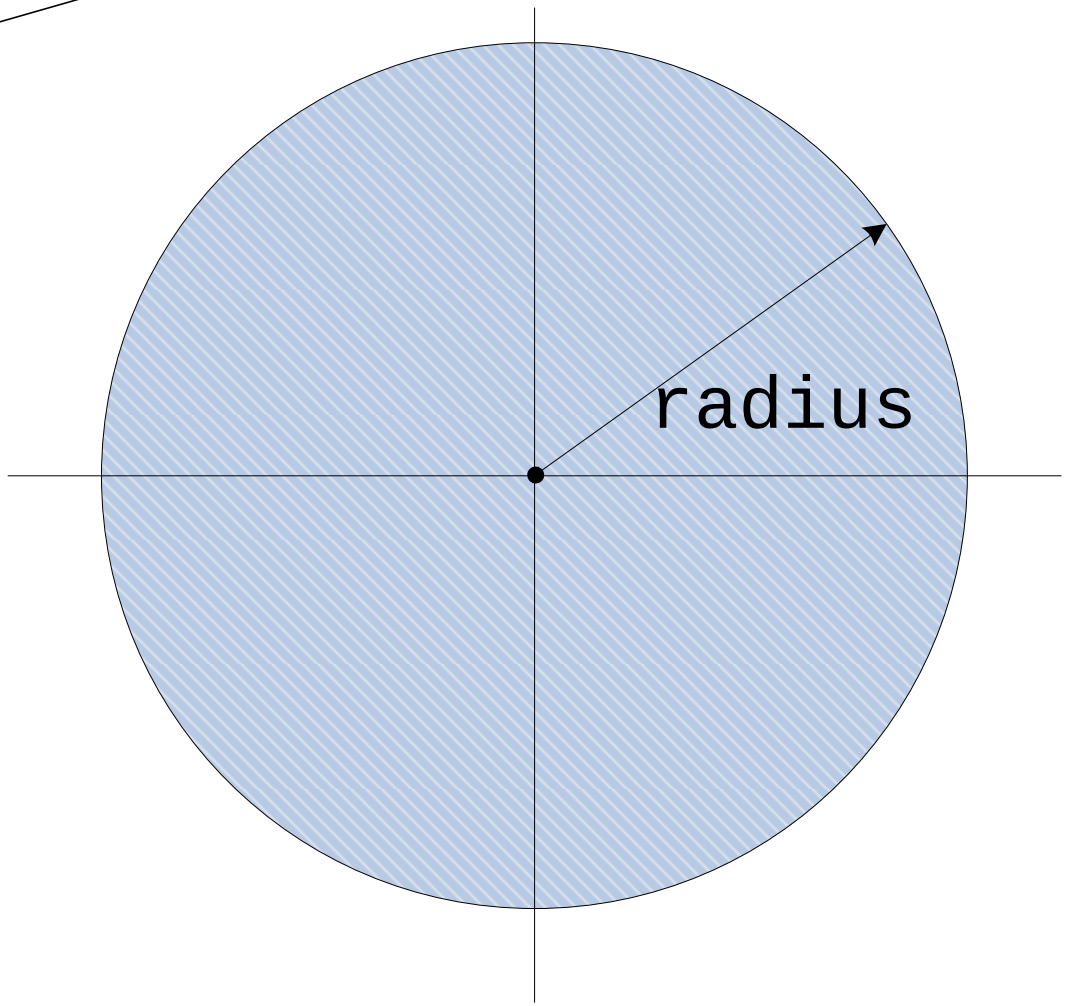
Enter, compile, and run a program, ***cp_ex02_area.c***
which can calculate a circle's area on your computer.

Requirements:
(1) Follow the **template** *cp_ex01_F2C.c,*
with detailed description,
*a **MAIN** function, call a sub-function*
*a **sub-function** for area calculation,*
cal_area()

(2) double double cal_area(double radius)
{
…
}

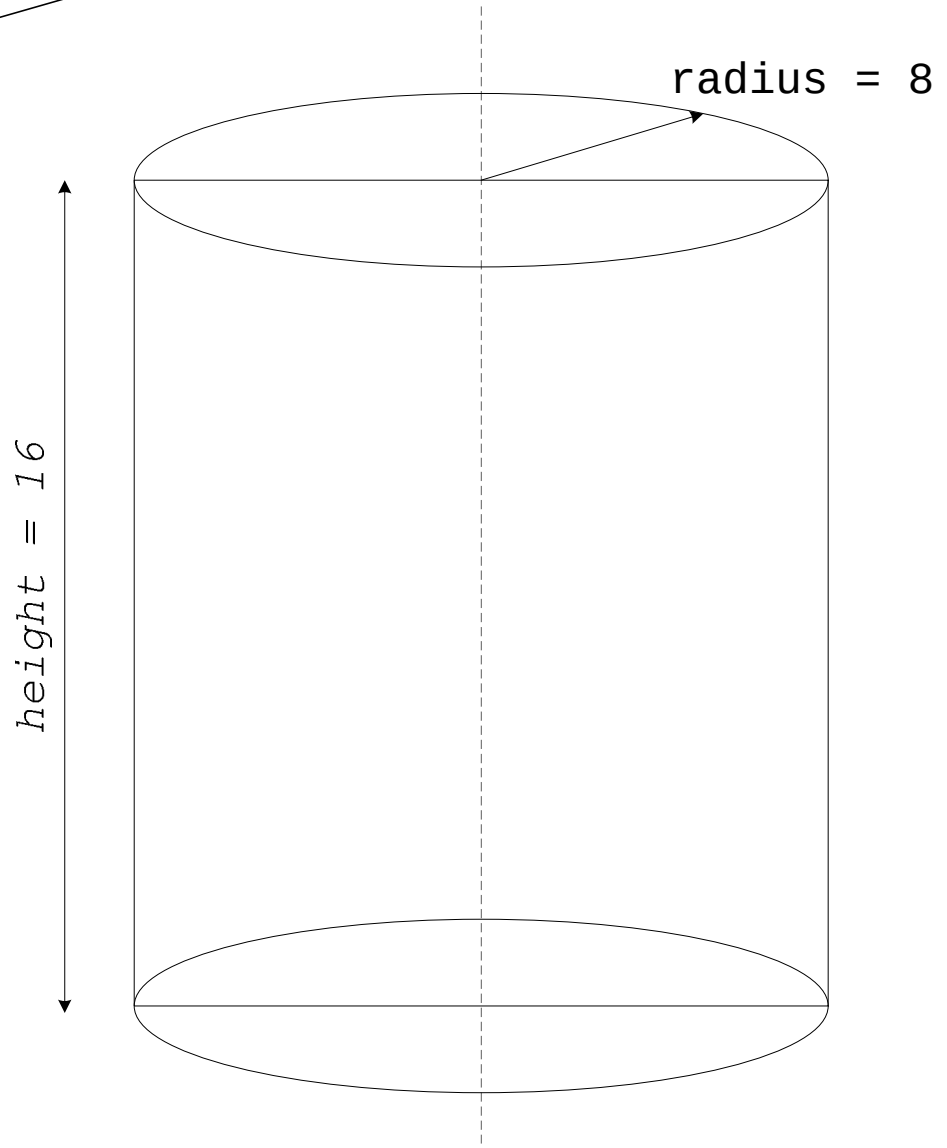
input: double radius
output: double area

Exercise-7



$$\text{Area} = \pi \ \text{radius}^2$$

Exercise-8



cp_ex03_volume.c

Determining the Volume of a Cylinder

Exercise-9

cp_ex04_PDF_normal.c

Consider the following programming problem: The equation of the normal (bell-shaped) curve used in statistical applications is:

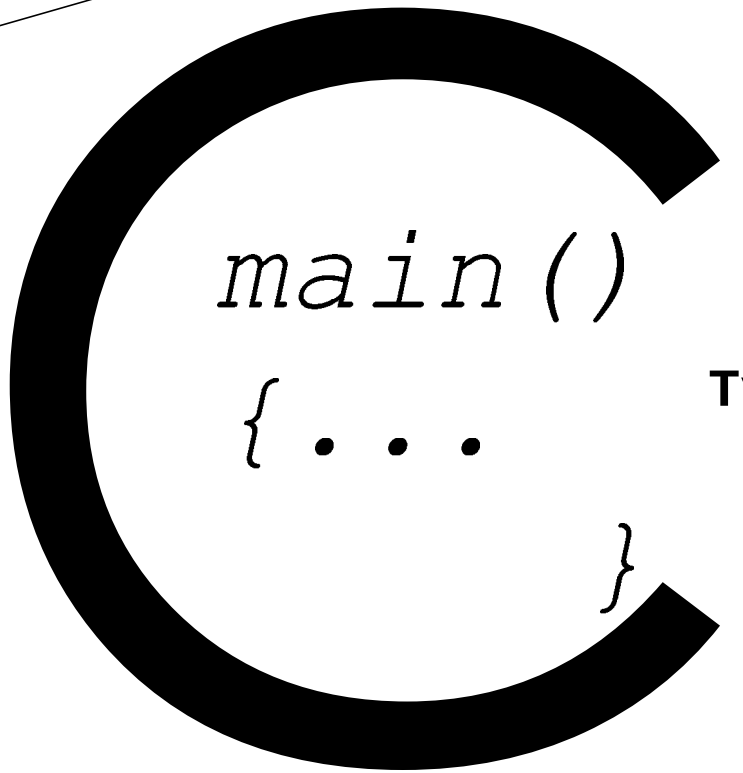
$$y = \frac{1}{\sigma\sqrt{2\pi}} e^{-[(½)(x-u)/\sigma]^2}$$

$$x \in [-1, 1]$$

using assuming,

```
mu      = 90,
sigma   = 4,
x       = 80,
e       = 2.71828,
pi      = 3.1416,
x_step  = 0.01
```

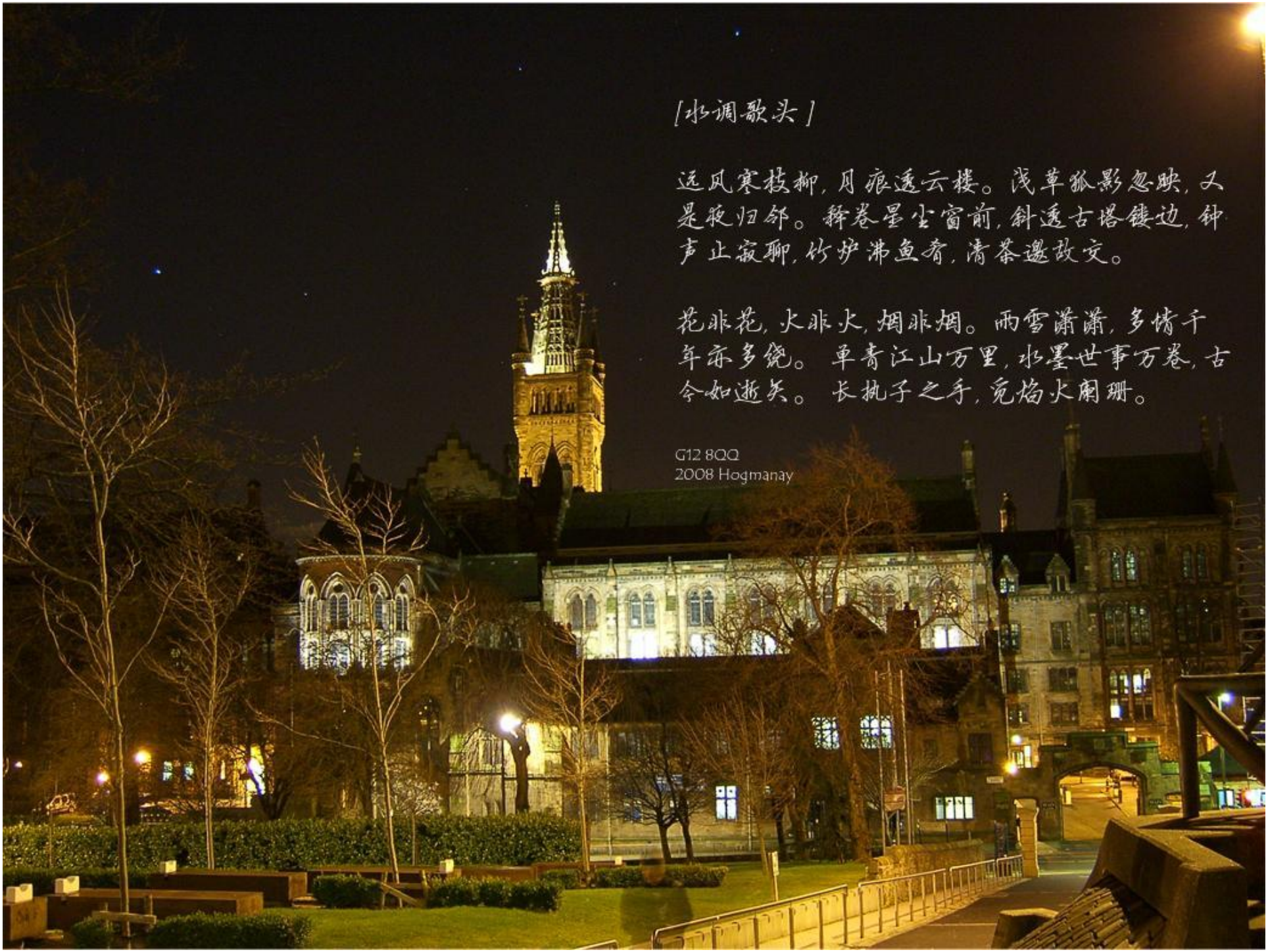
Chen , Yi
leo.chen.yi@live.co.uk
30-Jul-2010



Types, Operators and Expressions

End

BETA 1.0.0.1


[水调歌头]
远风寒枝柳，月痕透云楼。浅草孤影忽映，又是夜归邻。释卷墨尘窗前，斜透古塔镂边，钟声止寂聊，竹炉沸鱼肴，清茶邀故交。
花非花，火非火，烟非烟。雨雪潇潇，多情千年亦多绕。 单青江山万里，水墨世事万卷，古今如逝矣。 长执子之手，免焰火阑珊。
G12 8QQ
2008 Hogmanay