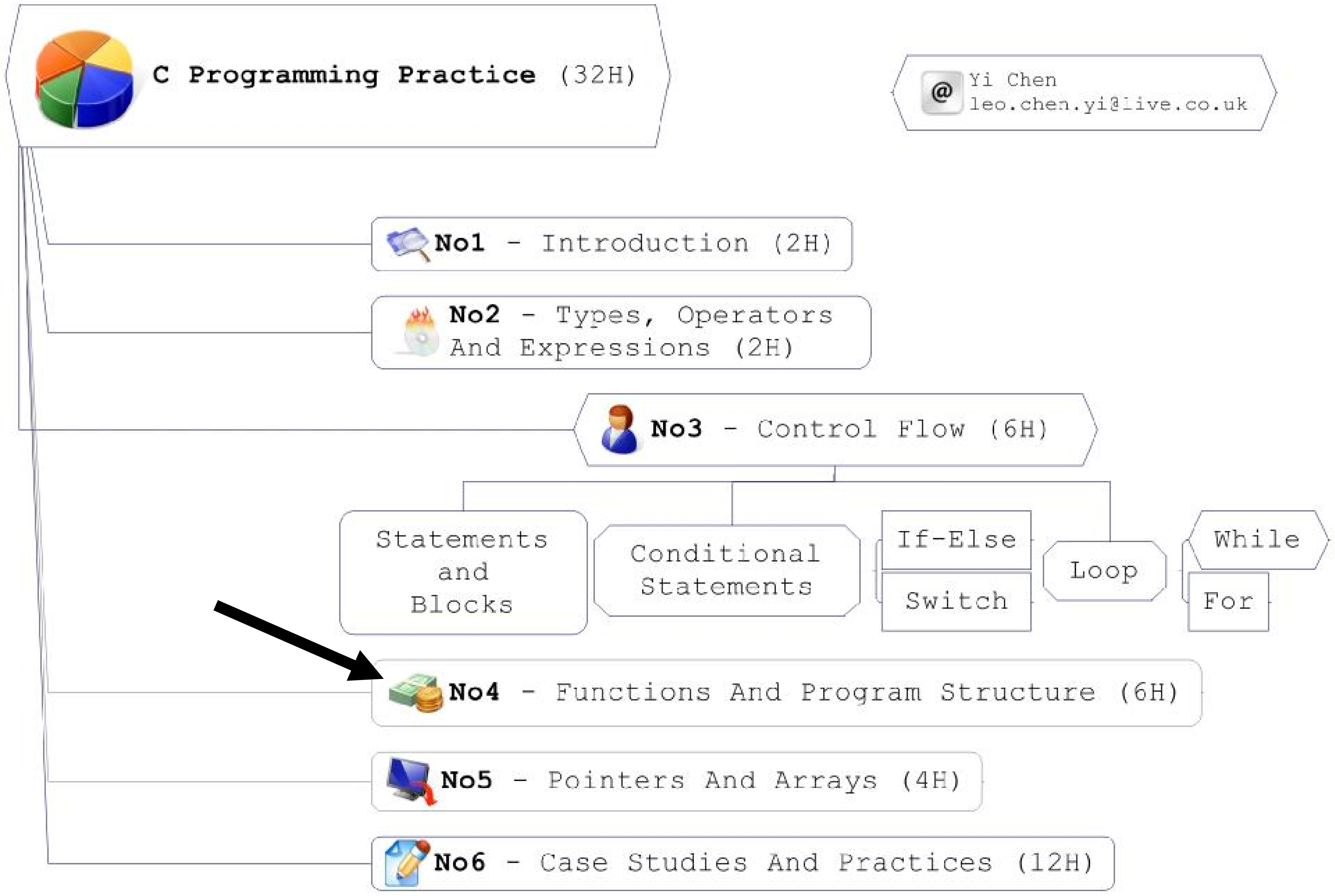
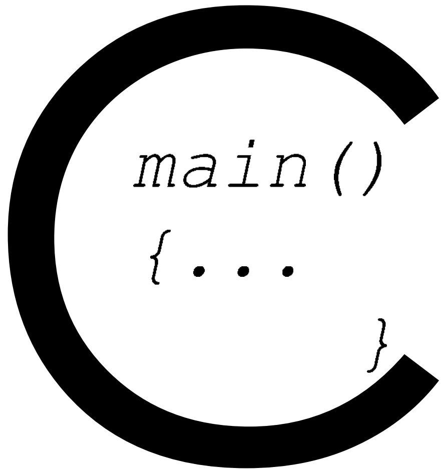

C Programming Practice (32H)
Yi Chen
leo.chen.yi@live.co.uk
No1 - Introduction (2H)
No2 - Types, Operators And Expressions (2H)
No3 - Control Flow (6H)
Statements and Blocks
Conditional Statements
If-Else
Switch
Loop
While
For
No4 - Functions And Program Structure (6H)
No5 - Pointers And Arrays (4H)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

C Programming Practice
No[4-1]

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



Functions and Program Structure

BETA 1.0.0.1

Contents

Basics of Functions
External Variables
Scope Rules
Header Files
Static Variables
Register Variables
Block Structure
Initialisation
Recursion
The C Preprocessor

Contents

à Basics of Functions
External Variables
Scope Rules
Header Files
Static Variables
Register Variables
Block Structure
Initialisation
Recursion
The C Preprocessor

Basics of Functions

C is a language of functions

The function name

Put your argument list here

Type of returned value

The function body

```
int  main( void )
   {
     printf( "Hello, World! \n" );

     getch();

     return 0;
   }
```

Basics of Functions

Structure of a Function

A general form of
a C function looks like this:

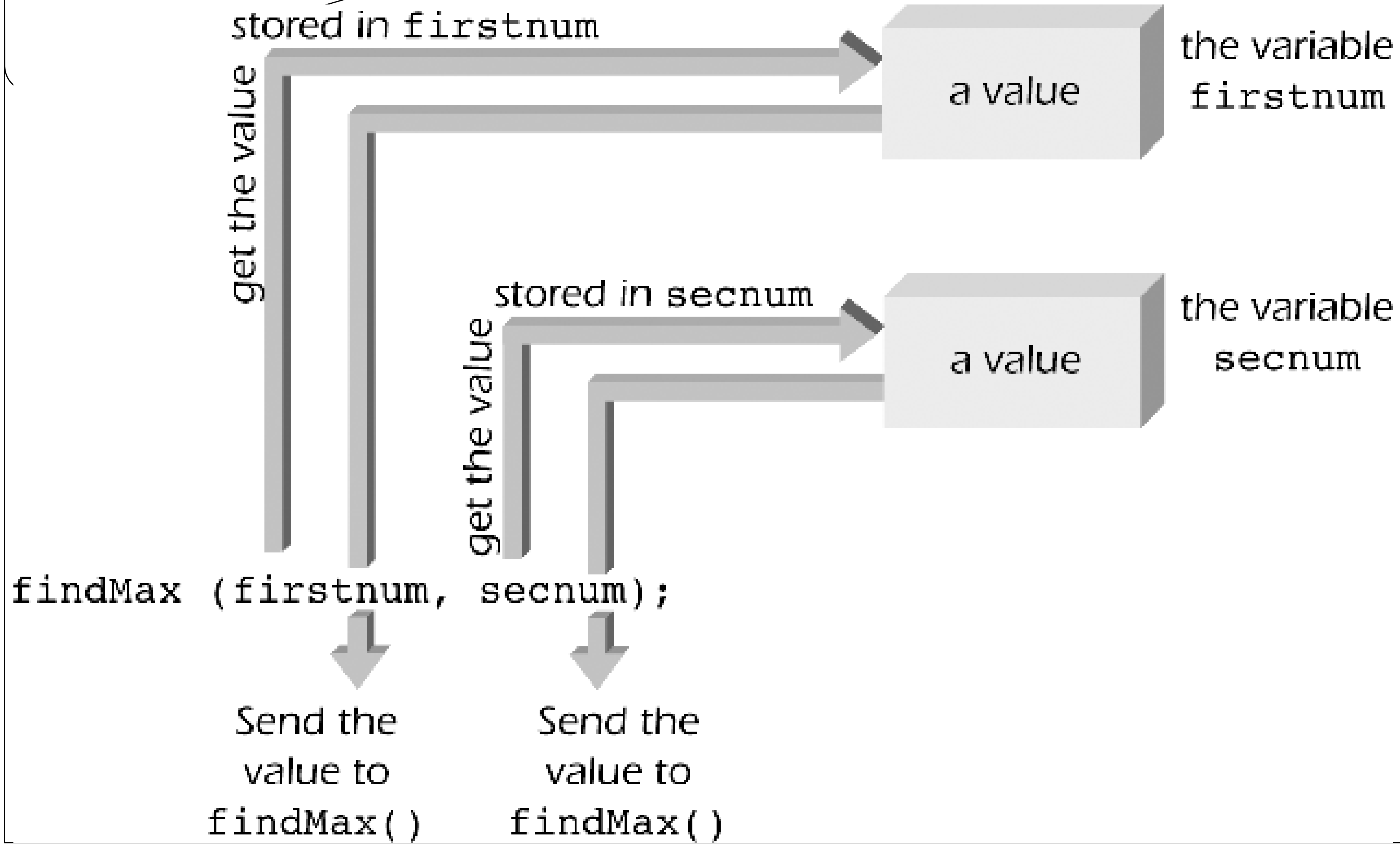
<*RETURN TYPE*> FUNCTIONNAME (ARGUMENT1, ARGUMENT2, ARGUMENT3......)
{
STATEMENT1;
STATEMENT2;
STATEMENT3; }

An example of function

```
int sum( int xdata, int  ydata )
{
   int result = 0;
       result = xdata + ydata;

 return result;            }
```

Basics of Functions



Basics of Functions

C is a language of functions

A function is a **group of statements** that together **perform a task**.

Additionally, at times we have used functions defined in a **standard library**, such as the ***pow*** function in the ***cmath*** library, used to raise a number to a certain power.

No program needs more than **ONE *main*** function. However, as you write more complex and sophisticated programs, you may find your *main* function becoming extremely long.

You need to **break** your long program into **sub-functions.**

Basics of Functions

C is a language of functions

1) Functions **break large** computing tasks **into smaller** ones;

2) **Re-use**, enable people to build on what others have done instead of starting over from scratch;

3) Appropriate functions **hide details** of operation from parts of the program that **don't need** to know about them, thus clarifying the whole, and easing the pain of making changes.

Basics of Functions

Modular Program: a program consisting of interrelated segments arranged in a logical and understandable form Easier to develop, correct, and modify than other kinds of programs.

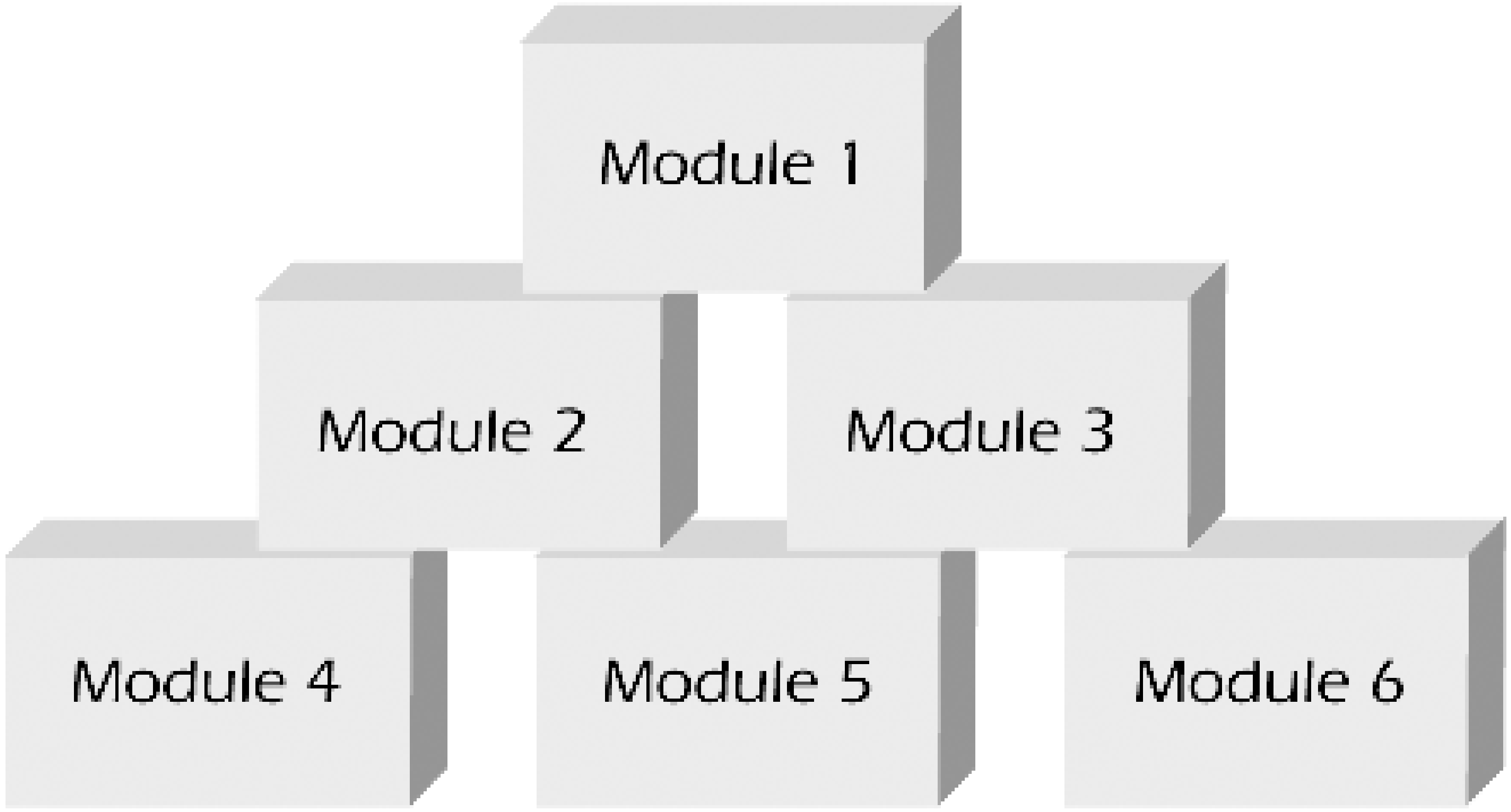
Module: a small segment which is designed to perform a specific task A group of modules is used to construct a modular program

C has been designed to make functions efficient and easy to use; C programs generally consist of **many small functions** rather than **a few big ones**.

A program may reside in **one** or **more** source files(.c).

Source files may be **compiled separately** and loaded together, along with previously compiled functions from libraries.

Basics of Functions



Basics of Functions

Types of functions

a function may belong to any one of
the following categories:

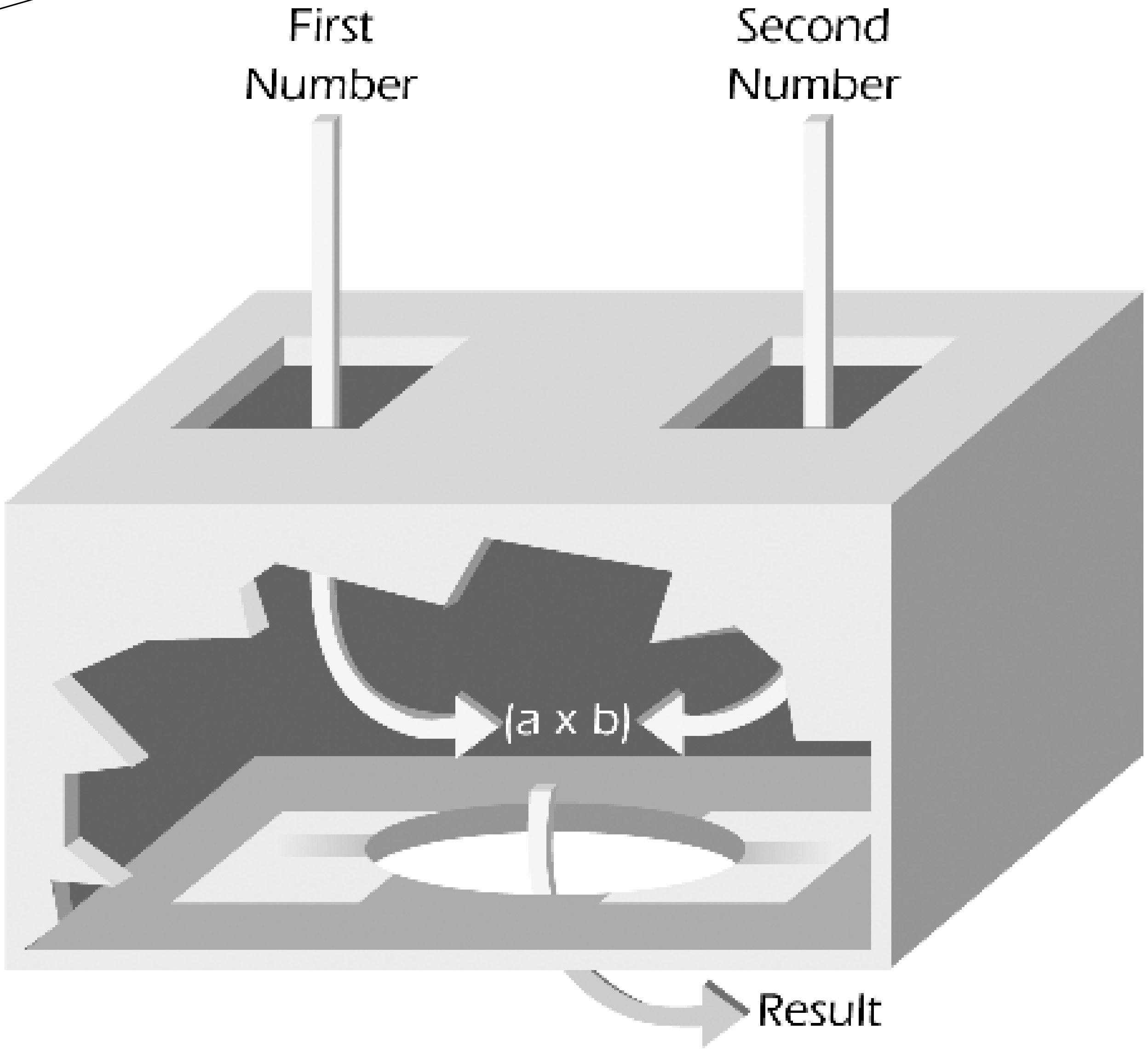
1) Functions with **no** arguments and **no** return values.

2) Functions with arguments and **no** return values.

3) Functions with **arguments** and return **values**.

4) Functions with **no** arguments and return **values**.

Basics of Functions



Contents

Basics of Functions
à External Variables
Scope Rules
Header Files
Static Variables
Register Variables
Block Structure
Initialization
Recursion
The C Preprocessor

External Variables

A C program consists of a set of **external objects**, which are either **variables** or **functions.**

The adjective ``external'' is used in contrast to ``**internal**'', which describes the arguments and variables defined **inside** functions.

External variables are defined **outside** of any function, and are thus potentially **available** to many **functions.**

Functions themselves are always **external**, because C does **not** allow functions to be **defined inside** other functions.

Initialisation of an external variable goes only with the definition.

External Variables

External variables are also useful because of their **greater scope and lifetime**.

Automatic variables are **internal** to a function; they come into existence when the function is entered, and disappear when it is left.

External variables, on the other hand, are permanent, so they can retain values from one function invocation to the next.

Thus if **two** functions must **share some** data, yet neither calls the other, it is often most convenient if the shared data is kept in **external variables** rather than being passed in and out via arguments.

```
 #define BUFSIZE 100

   char buf[BUFSIZE];     /* buffer for ungetch */
   int  bufp = 0;         /* next free position in buf
*/

   int getch(void)
{
       int flag = 0;

       return (bufp > 0) ? buf[--bufp] : getchar();
   }

   void ungetch(int cdata)
{
       if (bufp >= BUFSIZE)
           printf("ungetch: too many characters\n");
       else
           buf[bufp++] = cdata;
   }
```

Contents

Basics of Functions
External Variables
à Scope Rules
Header Files
Static Variables
Register Variables
Block Structure
Initialisation
Recursion
The C Preprocessor

Scope Rules

The functions and external variables that make up a C program need **not all be compiled** at the same time;

the source text of the program may be kept **in several files**, and previously compiled routines may be loaded from libraries. Thinking about: **how to organise .c and .h files**

How are **declarations** written so that variables are properly declared during **compilation**?

How are **declarations arranged** so that all the pieces will be properly **connected** when the program is loaded?

How are **declarations organised** so there is **only one copy**?

How are external variables **initialised**?

Contents

Basics of Functions
External Variables
Scope Rules
à Header Files
Static Variables
Register Variables
Block Structure
Initialisation
Recursion
The C Preprocessor

Header Files .h

To **centralise** the **definitions** and **declarations shared** among .c files.

So that there is **only one** copy to get and keep right as the program evolves.

Accordingly, we will place this common material in a *header file*, **.h**, which will be included as necessary.

The **#include** line is needed in a .c file when you need the function declaration included in the .h file.

For a much larger program, more organisation and more headers would be needed.

Contents

Basics of Functions
Functions Returning Non-integers
External Variables
Scope Rules
Header Files
à Static Variables
Register Variables
Block Structure
Initialisation
Recursion
The C Preprocessor

Static Variables

The **static** declaration, applied to an external variable or function, **limits** the scope of that object to the **rest** of the **source** file **being compiled**.

External static thus provides a way to **hide** names like ***buf*** and ***bufp*** in the ***getch-ungetch*** combination, which must be external so they can be **shared**, yet which should not be visible to users of **getch** and **ungetch**.

Static Variables

Static storage is specified by prefixing the normal declaration with the word *static*.

If the two routines and the two **variables** are **compiled** in **one** file, as in

```
  static char buf[BUFSIZE];  /* buffer for ungetch */
  static int bufp = 0; /* next free position in buf */

   int getch(void) { ... }

   void ungetch(int c) { ... }
```

Contents

Basics of Functions
Functions Returning Non-integers
External Variables
Scope Rules
Header Files
Static Variables
à Register Variables
Block Structure
Initialisation
Recursion
The C Preprocessor

Register Variables

A **register declaration** advises the **compiler** that the variable in question will be **heavily** used.

The idea is that **register variables** are to be **placed** in **machine registers**, which may result in **smaller** and **faster** programs. But **compilers** are free to ignore the advice.

The register declaration looks like:

```
register int  xdata = 0;
register char flag  = '';
```

In practice, there are **restrictions** on register variables, reflecting the realities of **underlying hardware**.

Register Variables

The register declaration can only be applied to **automatic variables** and to the **formal parameters** of **a function**.

In this later case, it looks like:

```
type function(
     register unsigned column,
     register long     row )

{
           register int flag = 0;  ...  }
```

Only a few variables *in each function may be kept in registers, and* ***only certain types*** *are allowed, vary from* ***machine to machine****.*

Contents

Basics of Functions
Functions Returning Non-integers
External Variables
Scope Rules
Header Files
Static Variables
Register Variables
à Block Structure
Initialisation
Recursion
The C Preprocessor

Block Structure

C is **not** a **block-structured** language in the sense of Pascal or similar languages, because **functions** may **not** be **defined within** other functions.

On the other hand, **variables** can be defined in a block-structured fashion within a function.

Declarations of **variables** (including initialisations) may follow the **left brace** that introduces *any* compound statement, not just the one that begins a function.

Variables declared in this way hide any identically named variables in outer blocks, and remain in existence until the matching **right brace**.

Block Structure

```
 if ( num > 0 )
{
    int idx;  /* declare a new idx */

    for (idx = 0; idx < num; idx++)
           ...
  }
```

the scope of the variable idx is the ``true'' branch of the if; **this idx** is **unrelated** to **any idx outside** the block.

An automatic variable declared and initialized in a block is initialised each time the block is entered.

Block Structure

Automatic variables, including formal parameters, also **hide external** variables and functions of the same name. Given the declarations

```
int xdata;
int ydata;

F(double xdata)
{
    double ydata;
}
```

then **within** the function F, occurrences of **xdata** refer to the parameter, which is a **double**;

outside F, they refer to the external **int**. The same is true of the variable ydata.

As a matter of style, it's best to **avoid** variable **names** that conceal names in an outer scope; the potential for confusion and error is too great.

Contents

Basics of Functions
Functions Returning Non-integers
External Variables
Scope Rules
Header Files
Static Variables
Register Variables
Block Structure
à Initialisation
Recursion
The C Preprocessor

Initialisation

char array

```
[1] char *string = ″String Line″;

[2] char string [] = ‶String Line″;

[3] char buffer[80];

    sprintf( buffer,
            "An approximation of Pi is %f \n", M_PI );

/* the size of buffer[] need to be allocated properly
That is, it need greater than the string length we
need later */
```

Initialisation

char array

[4] snprintf

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

int main(void)
{
     char str[10] = {'\0'};

     snprintf(str, sizeof(str), "0123456789012345678");

     printf("str = %s \n", str);

     return 0;
}
```

Initialisation

```
[5] char string1[10];
      strcpy (string1,″String″) ;

/* = strcpy(string1[0],″string″); */

[6] Static char name[2][8] = {‶Leo″,″Alan″ };
/* results: name[0] = ‶Leo″ , name[1] = ‶Alan ″*/

[7] char name[3] ;
      Name[0] = ‵ L ‵;
      Name[1] = ‵ e ‵;
      Name[2] = ‵ o ‵; /*result : name[0]=″leo″*/
```

Initialisation

int/double array

[1] double data [2]= { 1.2 , 2.3 };

```
/*= double data [ ]= { 1.2 , 2.3 }; */
```

[2] double data [2] ;

```
       data[0] = 1.2;
       data[1] = 2.3;
```

[3] int a[5] ={0 ,0 , 0, 0, 0};

```
   /*int a[5] = { 0 } ;*/
```

[4] if the length of array is un-known, see ***Dynamic Memory Allocation*** in ***Chapter: Pointer***

Initialisation

structure

```
struct student
{
 char name[12];
 char sex;
 int score;
} student1[2], *student1;

[1] Student1[2]= {
           { “leo”,  ”M”, ”88” },
           { “fredo”,”M”, ”90” }  };

[2] student1 = {0};
 /*only for every elements can be set to 0.
if don’t, you have to make a function of
init_fun( ) to initialise this structure */
```

Contents

Basics of Functions
Functions Returning Non-integers
External Variables
Scope Rules
Header Files
Static Variables
Register Variables
Block Structure
Initialisation
à Recursion
The C Preprocessor

Recursion

C functions may be used **recursively**; that is, a function may **call itself** either **directly** or **indirectly**.

When a function calls itself recursively, each invocation gets a **fresh** set of all the automatic variables, independent of the previous set.

Recursion

This program takes a small positive or zero integer value and **computes** the **factorial** for that number recursively and displays the result.

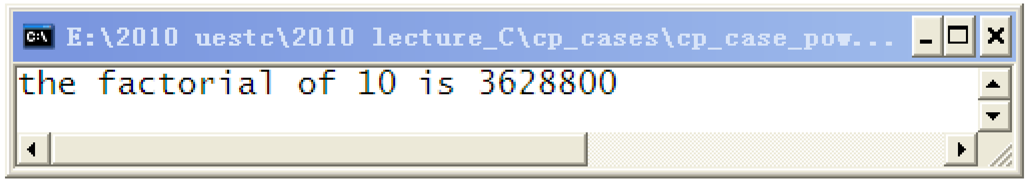
```
 int  main( void )
{
   int number = 10;

   printf("the factorial of %d is %d \n",
           number, factorial(number) );

   getch();

   return 0;              }
```

Recursion


```
E:\2010 uestc\2010 lecture_C\cp_cases\cp_case_pow...
the factorial of 10 is 3628800
```

```
int factorial(int number)
{
      int temp = 0;

      if( number <= 1 ) return 1;

      temp = number * factorial( number - 1 );

      return temp;    }
```

Contents

Basics of Functions
Functions Returning Non-integers
External Variables
Scope Rules
Header Files
Static Variables
Register Variables
Block Structure
Initialisation
Recursion
à The C Preprocessor

The C Preprocessor

C provides certain language facilities by means of a preprocessor, which is conceptionally a separate **first step** in **compilation**.

The **two** most frequently used features are **#include**, to **include** the contents of a **file** during compilation, and **#define**, to **replace** a **token** by an arbitrary sequence of characters.

Other features described in this section include conditional compilation and macros with arguments.

File Inclusion

The C Preprocessor

Any source line of the form **#include "*filename*"** or **#include <*filename*>** is replaced by the contents of the file *filename*.

If the *filename* is quoted, **searching** for the file typically **begins** where the source program was found; (same path as this .c file)

if it is **not** found there, or if the name is enclosed in **<** and **>**, searching follows an **implementation-defined rule** to find the file. (system path)

An included file may **itself** contain #include lines.

File Inclusion

#include is the preferred way to tie the declarations together for a **large program**.

It guarantees that all the source files will be **supplied** with the **same definitions** and variable declarations, and thus eliminates a particularly nasty kind of bug.

Naturally, when an included file is changed, **all files** that **depend** on it **must be recompiled**.

Macro Substitution

A definition has the form:

#define *name replacement text* **(include spaces)**

Subsequent occurrences of the token name will be replaced by the *replacement text*.

Normally the replacement text is the rest of the line, but a long definition may be continued onto several lines by placing a \ at the end of each line to be continued.

The scope of a name defined with #define is **from its point of definition** to the end of the source file being compiled.

Substitutions are made **only** for tokens, and do **not** take place **within quoted strings**.

The C Preprocessor

the replacement text is arbitrary

For example, if **YES** is a defined name, there would be **no substitution** in printf(**"YES"**) or in **YESMAN**.

define macros with **arguments**

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Thus the line

```
x = max(p+q, r+s);
```

```
max(i++, j++) /* WRONG */
```

will be replaced by the line

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Macro Substitution

Names may be **undefined** with **#undef**,

usually to **ensure** that a routine is really a function, not a macro:

```
#undef getchar

int getchar(void)
{
...

       }
```

Macro Substitution

a **parameter name** is preceded by a # in the replacement text, the combination will be expanded into a **quoted string** with the parameter replaced by the actual argument.

This can be combined with **string** concatenation to make, for example, a debugging print macro:

*#define dprint(expr) printf(**#expr** " = %g\n", expr)*

When this is invoked, as in *dprint(x/y)* the macro is expanded into *printf(**"x/y"** " = &g\n", x/y)*

Conditional Inclusion

It is possible to **control preprocessing** itself with **conditional statements** that are evaluated during preprocessing.

This provides a way to include code **selectively**, depending on the value of conditions evaluated during compilation.

The **#ifdef** and **#ifndef** lines are specialised forms that test whether a name is defined.

The first example of **#if** above could have been written

```
#if !defined(HDR)                 #ifndef HDR
#define HDR                       #define HDR
/* contents of hdr.h go here */ /* contents of hdr.h go here */
 #endif                           #endif
```

Conditional Inclusion

This sequence tests the name SYSTEM to decide which version of a header to include:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

The C Preprocessor

If you want to support both ANSI C and K&R C , you can use the following construction

```
#ifdef __STDC__
 /* ANSI code */
#else
 /* K and R code */
#endif
```

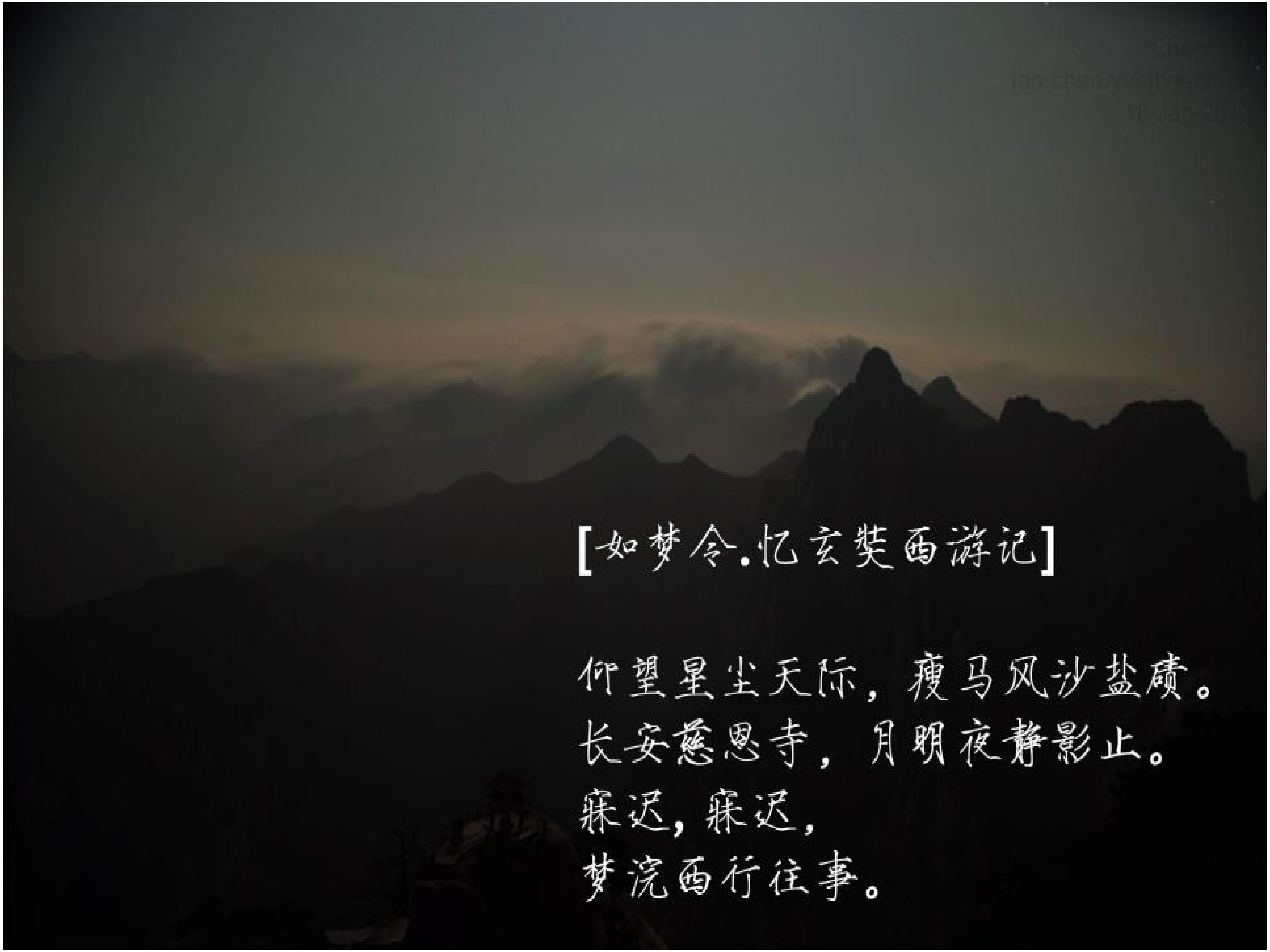
Chen , Yi
leo.chen.yi@live.co.uk
30-Jul-2010

main()
{...
}

Functions and Program Structure

End

BETA 1.0.0.1



[如梦令.忆玄奘西游记]

仰望星尘天际，瘦马风沙盐碛。
长安慈恩寺，月明夜静影止。
寐迟，寐迟，
梦浣西行往事。