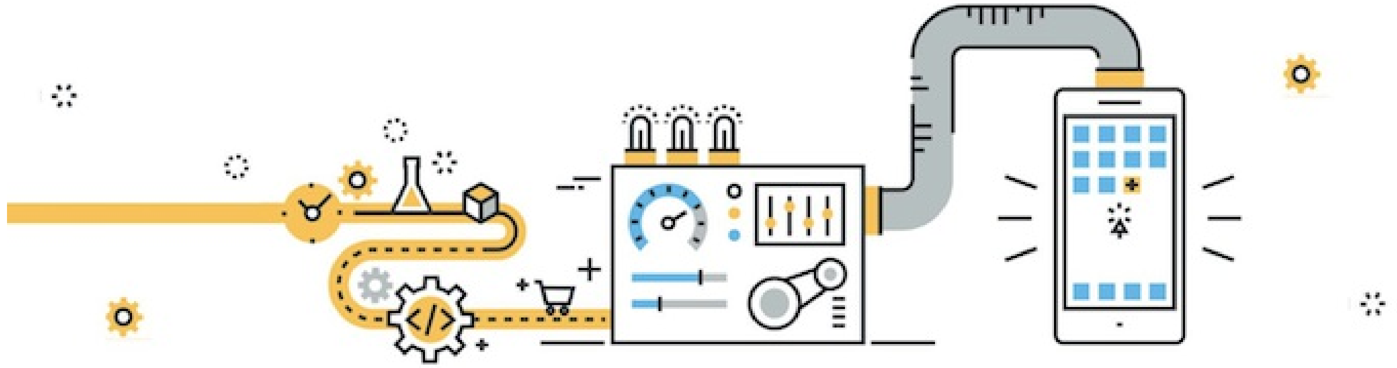
COMP9024 17s2

Data Structures and Algorithms



Michael Thielscher

Web Site: webcms3.cse.unsw.edu.au/COMP9024/17s2/

Course Convenor

Name:	Michael Thielscher
Office:	K17-401J (turn left from lift and dial 57129)
Phone:	9385 7129
Email:	mit@unsw.edu.au
Consults:	Thu 5-6pm, Forum
Research:	Artificial Intelligence, Robotics, General Problem-Solving Systems
Pastimes:	Fiction, Films, Food, Football

Course Convenor (cont)

Tutor: Shanush Prema Thasarathan,
shanushp@cse.unsw.edu.au

Tuesday, 2-4pm CSE Clavier Lab (LG20 in K14)

Course Goals

COMP9021 …

- gets you thinking like a programmer
- solving problems by developing programs
- expressing your ideas in the language Python

COMP9024 …

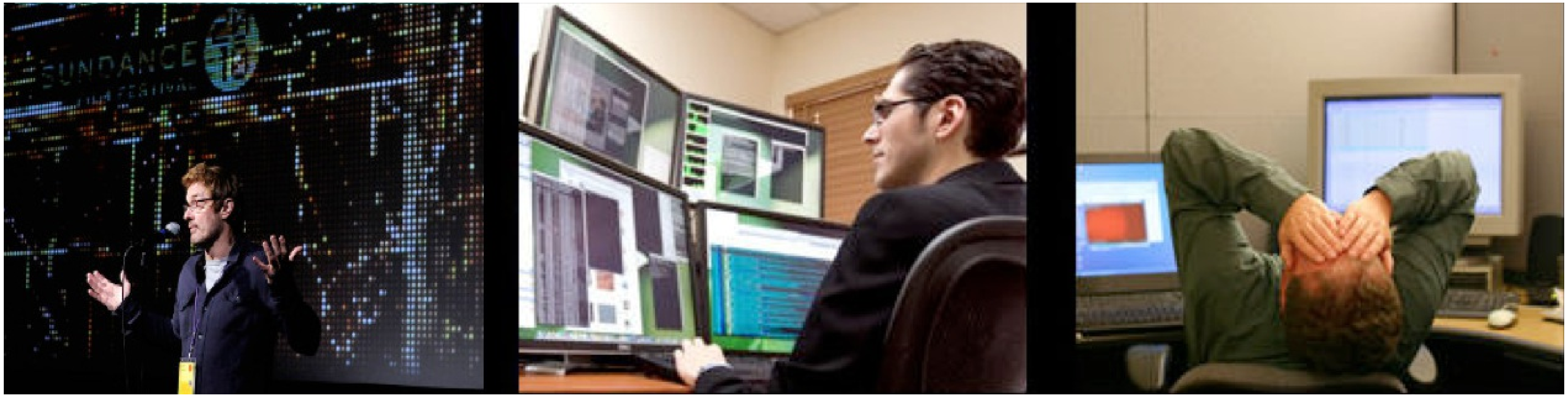
- gets you thinking like a computer scientist
- knowing fundamental data structures/algorithms
- able to reason about their applicability/effectiveness
- able to analyse the efficiency of programs
- able to code in C

Course Goals (cont)

COMP9021 …

Course Goals (cont)

COMP9024 …



Pre-conditions

At the *start* of this course you should be able to:

- produce correct programs from a specification
- understand the state-based model of computation (variables, assignment, function parameters)
- use fundamental data structures (characters, numbers, strings, arrays, linked lists, binary trees)
- use fundamental control structures (`if`, `while`, `for`)
- fix simple bugs in incorrect programs

Post-conditions

At the *end* of this course you should be able to:

- choose/develop effective data structures (DS)
- analyse performance characteristics of algorithms
- choose/develop algorithms (A) on these DS
- package a set of DS+A as an abstract data type
- develop and maintain C programs

COMP9024 Themes

Major themes …

1. Data structures, e.g. for graphs, trees
2. A variety of algorithms, e.g. on graphs, trees, strings
3. Analysis of algorithms

For data types: alternative data structures and implementation of operations

For algorithms: complexity analysis

Access to Course Material

All course information is placed on the course website:

- webcms3.cse.unsw.edu.au/COMP9024/17s2/

Slides/Problem Sets are publicly readable.

If you want to post/submit, you need to login.

Schedule

Week	**Lectures**	Ch	Notes
01	Introduction, C language	M2-4,7-8	
02	Abstract data types (ADTs)	S4	first help lab
03	Dynamic data structures	M10	Assignment 1
04	Analysis of algorithms	S2	|
05	*Break*		|
06	Graph data structures	S17	due
07	Graph algorithms: graph search	S18	
08	Graph algorithms: spanning trees, minimal paths	S20-21	
09	**Mid-term exam**		Assignment 2
—	*Mid-semester break*		|
10	Tree algorithms: balanced trees	S12-13	|
11	Tree algorithms: splay-, AVL-, red-black trees	S13	|
12	Text processing algorithms	S15	due
13	Randomised algorithms	—	last help lab

Credits for Material

Always give credit when you use someone else's work.

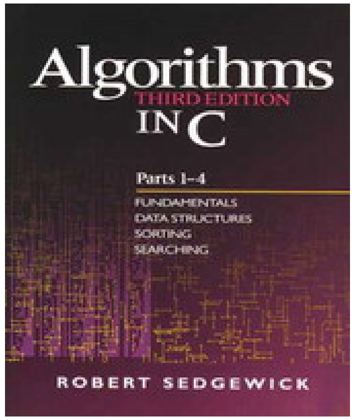
Ideas for the COMP9024 material are drawn from

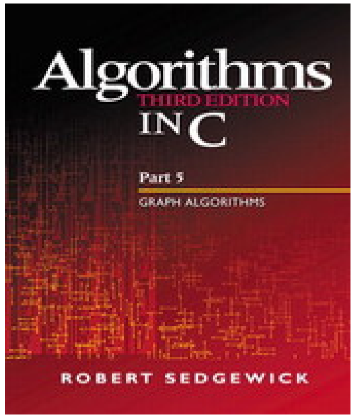
- slides by John Shepherd (COMP1927 16s2), Hui Wu (COMP9024 16s2) and Alan Blair (COMP1917 14s2)
- Robert Sedgewick's and Alistair Moffat's books

Resources

Textbook is a "double-header"

- Algorithms in C, Parts 1-4, Robert Sedgewick
- Algorithms in C, Part 5, Robert Sedgewick



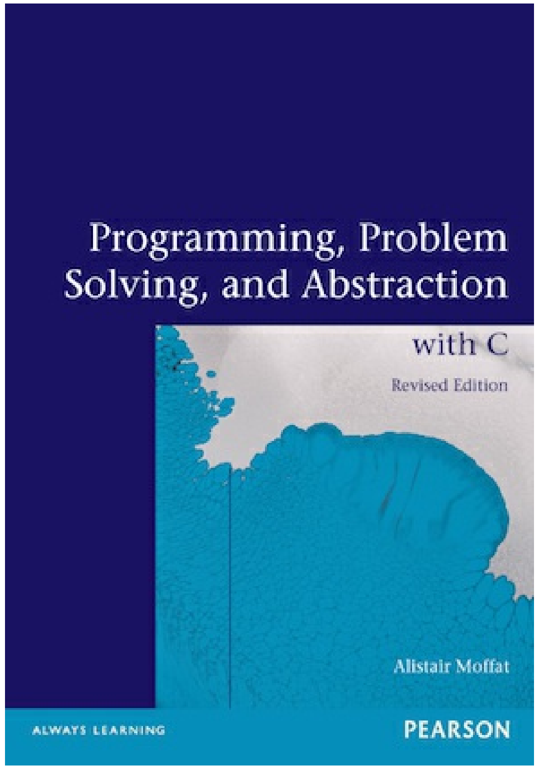


Good books, useful beyond COMP9024 (but coding style …)

Resources (cont)

Supplementary textbook:

- Alistair Moffat
  Programming, Problem Solving, and Abstraction with C
  Pearson Educational, Australia, Revised edition 2013, ISBN 978-1-48-601097-4



Also, numerous online C resources are available.

Lectures

Lectures will:

- present theory
- demonstrate problem-solving methods
- give practical demonstrations

Lectures provide an alternative view to textbook

Lecture slides will be made available before lecture

Feel free to ask questions, but No Idle Chatting

Problem Sets

The weekly homework aims to:

- clarify any problems with lecture material
- work through exercises related to lecture topics
- give practice with algorithm design skills (think before coding)

Problem sets available on web at the time of the lecture

Sample solutions will be posted in the following week

Do them yourself! and Don't fall behind!

Assignments

The assignments give you experience applying tools/techniques
(but to a larger programming problem than the homework)

The assignments will be carried out individually.

Both assignments will have a deadline at 11:59pm.

15% penalty will be applied to the maximum mark for every 24 hours late after the deadline.

- 1 day late: mark is capped to 85% of the maximum possible mark
- 2 days late: mark is capped to 70% of the maximum possible mark
- 3 days late: mark is capped to 55% of the maximum possible mark
- …

The two assignments contribute 10% + 15% to overall mark.

Assignments (cont)

Advice on doing the assignments:

They always take longer than you expect.

Don't leave them to the last minute.

Organising your time → no late penalty.

If you do leave them to the last minute:

- take the late penalty rather than copying

Plagiarism

Just Don't Do it

We get very annoyed by people who plagiarise.

Plagiarism will be checked for and punished.

Help Lab

The help lab:

- aims to help you if you have difficulties with the weekly programming exercises
- … and the assignments
- non-programming exercises from problem sets may also be discussed

Tuesdays (Week 2-13) from 2-4pm in CSE Clavier Lab (LG20, Bldg K14) (walk past Keith Burrows (J14) towards Old Main)

Attendance is entirely voluntary

Exams

1-hour written mid-term exam in week 9 (21 September). Format:

- some multiple-choice questions
- some descriptive/analytical questions

2-hour ~~torture~~ written exam during the exam period. Format:

- some multiple-choice questions
- some descriptive/analytical questions

Exams (cont)

How to pass the Exams:

- do the Homework yourself
- do the Homework every week
- do the Assignments yourself
- practise programming outside classes
- read the lecture notes
- read the corresponding chapters in the textbooks

Assessment Summary

```
ass1  = mark for assignment 1   (out of 10)
ass2  = mark for assignment 2   (out of 15)
mid   = mark for mid-term exam  (out of 25)
final = mark for final exam     (out of 50)

if (mid+final >= 35)
   total = ass1 + ass2 + mid + final
else
   total = (mid+final) / 0.75;
```

To pass the course, you must achieve:

- at least 35/75 for **`mid+final`**
- at least 50/100 for **`total`**

Summary

The goal is for you to become a better Computer Scientist

- more confident in your own ability to choose data structures
- more confident in your own ability to develop algorithms
- able to analyse and justify your choices
- producing a better end-product
- ultimately, enjoying the program design process

C Programming Language

Why C?

- good example of an imperative language
- gives the programmer great control
- produces fast code
- many libraries and resources
- widely used in industry (and science)

Brief History of C

- C and UNIX opearting system share a complex history
- C was originally designed for and implemented on UNIX on a PDP-11 computer
- Dennis Ritchie was the author of C (around 1971)
- In 1973, UNIX was rewritten in C
- B (author: Ken Thompson, 1970) was the predecessor to C, but there was no A

Brief History of C (cont)

- B was a typeless language
- C is a typed language
- In 1983, American National Standards Institute (ANSI) established a committee to clean up and standardise the language
- ANSI C standard published in 1988
  - this greatly improved source code portability
- C is the main language for writing operating systems and compilers; and is commonly used for a variety of applications

Basic Structure of a C Program

```
// include files
// global definitions

// function definitions
function_type f(arguments) {

   // local variables

   // body of function

  return …;
}

.
.
```

```
.
.
.
.
.

// main function
int main(arguments) {

   // local variables

   // body of main function

   return 0;
}
```

Exercise #1: What does this program compute?

```
#include <stdio.h>

int f(int m, int n) {

   while (m != n) {
      if (m > n) {
  m = m-n;
      } else {
  n = n-m;
      }
   }
   return m;
}

int main(void) {

   printf("%d\n", f(30,18));
   return 0;
}
```

Example: Insertion Sort in C

Reminder — Insertion Sort algorithm:

```
insertionSort(A):
|   Input array A[0..n-1] of n elements
|
|   for all i=1..n-1 do
|   |   element=A[i], j=i-1
|   |   while j≥0 ∧ A[j]>element do
|   |       A[j+1]=A[j]
|   |       j=j-1
|   |   end while
|   |   A[j+1]=element
|   end for
```

Example: Insertion Sort in C (cont)

```c
#include <stdio.h>

#define SIZE 6

void insertionSort(int array[], int n) {
   int i;
   for (i = 1; i < n; i++) {
      int element = array[i];                  // for this element ...
      int j = i-1;
      while (j >= 0 && array[j] > element) {   // ... work down the ordered list
         array[j+1] = array[j];                // ... moving elements up
         j--;
      }
      array[j+1] = element;                    // and insert in correct position
   }
}

int main(void) {
   int numbers[SIZE] = { 3, 6, 5, 2, 4, 1 };
   int i;

   insertionSort(numbers, SIZE);
   for (i = 0; i < SIZE; i++)
      printf("%d\n", numbers[i]);

   return 0;
}
```

Example: Insertion Sort in C (cont)

```
#include <stdio.h> // include standard I/O library defs and functions

#define SIZE 6     // define a symbolic constant

void insertionSort(int array[], int n) {  // function headers must provide types
   int i;                                 // each variable must have a type
   for (i = 1; i < n; i++) {              // for-loop syntax
      int element = array[i];
      int j = i-1;
      while (j >= 0 && array[j] > element) {   // logical AND
         array[j+1] = array[j];
         j--;                                  // abbreviated assignment j=j-1
      }
      array[j+1] = element;                    // statements terminated by ;
   }                                           // code blocks enclosed in { }
}

int main(void) {                               // main: program starts here
   int numbers[SIZE] = { 3, 6, 5, 2, 4, 1 };   /* array declaration
                                                  and initialisation */
   int i;
   insertionSort(numbers, SIZE);
   for (i = 0; i < SIZE; i++)
      printf("%d\n", numbers[i]);              // printf defined in <stdio>

   return 0;           // return program status (here: no error) to environment
}
```

Compiling with `gcc`

$$\begin{tikzcd}\text{C source code: } \texttt{prog.c} \arrow[d] \\ \texttt{a.out} \quad \text{(executable program)}\end{tikzcd}$$

To compile a program **`prog.c`**, you type the following:

```
prompt$ gcc prog.c
```

To run the program, type:

```
prompt$ ./a.out
```

Compiling with **gcc** (cont)

Command line options:

- The default with **gcc** is not to give you any warnings about potential problems
- Good practice is to be tough on yourself:

```
prompt$ gcc -Wall prog.c
```

  which reports all warnings to anything it finds that is potentially wrong or non ANSI compliant

- The **-o** option tells **gcc** to place the compiled object in the named file rather than **a.out**

```
prompt$ gcc -o prog prog.c
```

Algorithms in C

Basic Elements

Algorithms are built using

- assignments
- conditionals
- loops
- function calls/return statements

Assignments

- In C, each statement is terminated by a semicolon `;`
- Curly brackets `{ }` used to enclose statements in a block
- The operators `++` and `--` can be used to increment a variable (add 1) or decrement a variable (subtract 1)
  - It is recommended to put the increment or decrement operator after the variable:

```
           // suppose k=6 initially
k++;       // increment k by 1; afterwards, k=7
n = k--;   // first assign k to n, then decrement k by 1
           // afterwards, k=6 but n=7
```

  - It is also possible (but NOT recommended) to put the operator before the variable:

```
           // again, suppose k=6 initially
++k;       // increment k by 1; afterwards, k=7
n = --k;   // first decrement k by 1, then assign k to n
           // afterwards, k=6 and n=6
```

Assignments (cont)

C assignment statements are really expressions

- they return a result: the value being assigned
- the return value is generally ignored

Frequently, assignment is used in loop continuation tests

- to combine the test with collecting the next value
- to make the expression of such loops more concise

Example: The pattern

```
v = getNextItem();
while (v != 0) {
    process(v);
    v = getNextItem();
}
```

can be written as

```
while ((v = getNextItem()) != 0) {
    process(v);
}
```

Exercise #2: What are the final values of `a` and `b`?

1.

```
a = 1; b = 7;
while (a < b) {
    a++;
    b--;
}
```

2.

```
a = 1; b = 5;
while ((a += 2) < b) {
    b--;
}
```

1. `a == 4, b == 4`
2. `a == 5, b == 4`

Conditionals

```
if (expression) {
    some statements;
}

if (expression) {
    some statements1;
} else {
    some statements2;
}
```

- ***some statements*** executed if, and only if, the evaluation of ***expression*** is non-zero
- ***some statements***$_1$ executed when the evaluation of ***expression*** is non-zero
- ***some statements***$_2$ executed when the evaluation of ***expression*** is zero
- Statements can be single instructions or blocks enclosed in **{ }**

Conditionals (cont)

Indentation is very important in promoting the readability of the code

Each logical block of code is indented:

```
// Style 1
if (x)
{
    statements;
}
```

```
// Style 2 (preferred)
if (x) {
    statements;
}
```

```
// Preferred else-if style
if (expression1) {
    statements1;
} else if (exp2) {
    statements2;
} else if (exp3) {
    statements3;
} else {
    statements4;
}
```

Conditionals (cont)

Relational and logical operators

`a > b`	**a** greater than **b**
`a >= b`	**a** greater than or equal **b**
`a < b`	**a** less than **b**
`a <= b`	**a** less than or equal **b**
`a == b`	**a** equal to **b**
`a != b`	**a** not equal to **b**
`a && b`	**a** logical and **b**
`a || b`	**a** logical or **b**
`! a`	logical not **a**

A relational or logical expression evaluates to `1` if true, and to `0` if false

Exercise #3: Conditionals

1. What is the output of the following program?

```
if ((x > y) && !(y-x <= 0)) {
    printf("Aye\n");
} else {
    printf("Nay\n");
}
```

2. What is the resulting value of **`x`** after the following assignment?

```
x = (x >= 0) + (x < 0);
```

1. The condition is unsatisfiable, hence the output will always be

```
Nay
```

2. No matter what the value of **`x`**, one of the conditions will be true (==**`1`**) and the other false (==**`0`**)
   Hence the resulting value will be **`x == 1`**

Sidetrack: Printing Variable Values with `printf()`

Formatted output written to standard output (e.g. screen)

```
printf(format-string, expr1, expr2, …);
```

format-string can use the following placeholders:

%d	decimal	**%f**	fixed-point
%c	character	**%s**	string
\n	new line	**\"**	quotation mark

Examples:

```
num = 3;
printf("The cube of %d is %d.\n", num, num*num*num);
```

```
The cube of 3 is 27.
```

```
char id  = 'z';
int  num = 1234567;
printf("Your \"login ID\" will be in the form of %c%d.\n", id, num);
```

```
Your "login ID" will be in the form of z1234567.
```

- Can also use width and precision:

```
printf("%8.3f\n", 3.14159);
```

```
   3.142
```

Loops

C has two different "while loop" constructs

```
// while loop
while (expression) {
     some statements;
}
```

```
// do .. while loop
do {
    some statements;
} while (expression);
```

The **`do .. while`** loop ensures the statements will be executed at least once

Loops (cont)

The "for loop" in C

```
for (expr1; expr2; expr3) {
    some statements;
}
```

- **`expr1`** is evaluated before the loop starts
- **`expr2`** is evaluated at the beginning of each loop
  - if it is non-zero, the loop is repeated
- **`expr3`** is evaluated at the end of each loop

Example:

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

Exercise #4: What is the output of this program?

```
int i, j;
for (i = 8; i > 1; i /= 2) {
    for (j = i; j >= 1; j--) {
        printf("%d%d\n", i, j);
    }
    putchar('\n');
}
```

```
88
87
..
81

43
..
41

22
21
```

Functions

Functions have the form

```
return-type function-name(parameters) {
    declarations
    statements
    return …;
}
```

- if ***return_type*** is **void** then the function does not return a value
- if ***parameters*** is **void** then the function has no arguments

Functions (cont)

When a function is called:

1. memory is allocated for its parameters and local variables
2. the parameter expressions in the calling function are evaluated
3. C uses "call-by-value" parameter passing …
   - the function works only on its own local copies of the parameters, not the ones in the calling function
4. local variables need to be assigned before they are used (otherwise they will have "garbage" values)
5. function code is executed, until the first **`return`** statement is reached

Functions (cont)

When a **`return`** statement is executed, the function terminates:

```
return expression;
```

1. the returned ***`expression`*** will be evaluated
2. all local variables and parameters will be thrown away when the function terminates
3. the calling function is free to use the returned value, or to ignore it

Example:

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

The return statement can also be used to terminate a function of return-type **`void`**:

```
return;
```

C Style Guide

UNSW Computing provides a style guide for C programs:

C Coding Style Guide
(http://wiki.cse.unsw.edu.au/info/CoreCourses/StyleGuide)

Not mandatory for COMP9024, but very useful guideline

- use proper layout, including indentation
- keep functios short and break into sub-functions as required
- use meaningful names (for variables, functions etc)

Sidetrack: Obfuscated Code

C has a reputation for allowing obscure code, leading to …

The International Obfuscated C Code Contest

- Run each year since 1984
- Goal is to produce
    - a working C program
    - whose appearance is obscure
    - whose functionality unfathomable
- Web site: **`www.ioccc.org`**
- 100's of examples of bizarre C code
  (understand these → you are a C master)

Sidetrack: Obfuscated Code (cont)

Most artistic code (Eric Marshall, 1986)

```
                                                   extern int
                                                       errno
                                                         ;char
                                                            grrr
                             ;main(                           r,
  argv, argc )            int    argc                           ,
   r        ;           char *argv[];{int                     P( );
#define x  int i,       j,cc[4];printf("     choo choo\n"     ) ;
x  ;if    (P(  !        i              )        |  cc[  !      j ]
&  P(j    )>2  ?        j              :        i  ){*  argv[i++ +!-i]
;              for    (i=              0;;    i++                   );
_exit(argv[argc- 2    / cc[1*argc]|-1<4 ]    ) ;printf("%d",P(""));}}
  P  (    a  )   char a   ;  {    a  ;   while(    a  >      "  B   "
  /* -    by E            ricM    arsh             all-      */);    }
```

Sidetrack: Obfuscated Code (cont)

Just plain obscure (Ed Lycklama, 1985)

```
#define o define
#o ___o write
#o ooo (unsigned)
#o o_o_ 1
#o _o_ char
#o _oo goto
#o _oo_ read
#o o_o for
#o o_ main
#o o__ if
#o oo_ 0
#o _o(_,__,___)(void)___o(_,__,ooo(___))
#o __o (o_o_<<((o_o_<<(o_o_<<o_o_))+(o_o_<<o_o_)))+(o_o_<<(o_o_<<(o_o_<<o_o_)))
o_(){_o_ _=oo_,__,___,____[__o];_oo ______;_____:___=__o-o_o_; _______:
_o(o_o_,____,__=(_-o_o_<___?_-o_o_:___));o_o(;__;_o(o_o_,"\b",o_o_),__--);
_o(o_o_," ",o_o_);o__(--___)_oo _______;_o(o_o_,"\n",o_o_);______:o__(_=_oo_(
oo_,____,__o))_oo _____;}
```

Data Structures in C

Basic Data Types

- In C each variable must have a type
- C has the following generic data types:

char	character	**'A'**, **'e'**, **'#'**, …
int	integer	**2**, **17**, **-5**, …
float	floating-point number	**3.14159**, …
double	double precision floating-point	**3.14159265358979**, …

There are other types, which are variations on these

- Variable declaration must specify a data type and a name; they can be initialised when they are declared:

```
float x;
char  ch = 'A';
int   j = i;
```

Symbolic Constants

We can define a symbolic constant at the top of the file

```
#define SPEED_OF_LIGHT 299792458.0
```

Symbolic constants used to avoid burying "magic numbers" in the code

Symbolic constants make the code easier to understand and maintain

```
#define NAME replacement_text
```

- The compiler's pre-processor will replace all occurrences of **`name`** with **`replacement_text`**
- it will **not** make the replacement if **`name`** is inside quotes (`"…"`) or part of another name

Example:
The constants **`TRUE`** and **`FALSE`** are often used when a condition with logical value is wanted.
They can be defined by:

```
#define TRUE  1
#define FALSE 0
```

Basic Aggregate Data Types

Aggregate Data Types

Families of aggregate data types:

- homogenous … all elements have same base type
  - arrays (e.g. **`char s[50]`**, **`int v[100]`**)
- heterogeneous … elements may combine different base types
  - structures

Arrays

An array is

- a collection of same-type variables
- arranged as a linear sequence
- accessed using an integer subscript
- for an array of size *N*, valid subscripts are 0..*N-1*

Examples:

```
int  a[20];     // array of 20 integer values/variables
char b[10];     // array of 10 character values/variables
```

Arrays (cont)

Larger example:

```
#define MAX 20

int i;              // integer value used as index
int fact[MAX];      // array of 20 integer values

fact[0] = 1;
for (i = 1; i < MAX; i++) {
    fact[i] = i * fact[i-1];
}
```

Strings

"String" is a special word for an array of characters

- end-of-string is denoted by `'\0'` (of type **`char`** and always implemented as 0)

Example:

If a character array **`s[11]`** contains the string **`"hello"`**, this is how it would look in memory:

```
  0    1    2    3    4    5    6    7    8    9    10
-----------------------------------------------------
| h  | e  | l  | l  | o  | \0|    |    |    |    |    |
-----------------------------------------------------
```

Array Initialisation

Arrays can be initialised by code, or you can specify an initial set of values in declaration.

Examples:

```
char s[6]    = {'h', 'e', 'l', 'l', 'o', '\0'};

char t[6]    = "hello";

int fib[20] = {1, 1};

int vec[]    = {5, 4, 3, 2, 1};
```

In the third case, **`fib[0] == fib[1] == 1`** while the initial values **`fib[2] .. fib[19]`** are undefined.

In the last case, C infers the array length (as if we declared **`vec[5]`**).

Exercise #5: What is the output of this program?

```c
#include <stdio.h>

int main(void) {
   int arr[3] = {10,10,10};
   char str[] = "Art";
   int i;

   for (i = 1; i < 3; i++) {
      arr[i] = arr[i-1] + arr[i] + 1;
      str[i] = str[i+1];
   }
   printf("Array[2] = %d\n", arr[2]);
   printf("String = \"%s\"\n", str);
   return 0;
}
```

```
Array[2] = 32
String = "At"
```

Arrays and Functions

When an array is passed as a parameter to a function

- the address of the start of the array is actually passed

Example:

```
int total, vec[20];
…
total = sum(vec);
```

Within the function …

- the types of elements in the array are known
- the size of the array is unknown

Arrays and Functions (cont)

Since functions do not know how large an array is:

- pass in the size of the array as an extra parameter, or
- include a "termination value" to mark the end of the array

So, the previous example would be more likely done as:

```
int total, vec[20];
…
total = sum(vec,20);
```

Also, since the function doesn't know the array size, it can't check whether we've written an invalid subscript (e.g. in the above example 100 or 20).

Exercise #6: Arrays and Functions

Implement a function that sums up all elements in an array.

Use the prototype

```
int sum(int[], int)
int sum(int vec[], int dim) {
   int i, total = 0;

   for (i = 0; i < dim; i++) {
      total += vec[i];
   }
   return total;
}
```

Multi-dimensional Arrays

Examples:

```
float q[2][2];
```

$$\begin{bmatrix} 0.5 & 2.7 \\ 3.1 & 0.1 \end{bmatrix}$$

```
int r[3][4];
```

$$\begin{bmatrix} 5 & 10 & -2 & 4 \\ 0 & 2 & 4 & 8 \\ 21 & 2 & 1 & 42 \end{bmatrix}$$

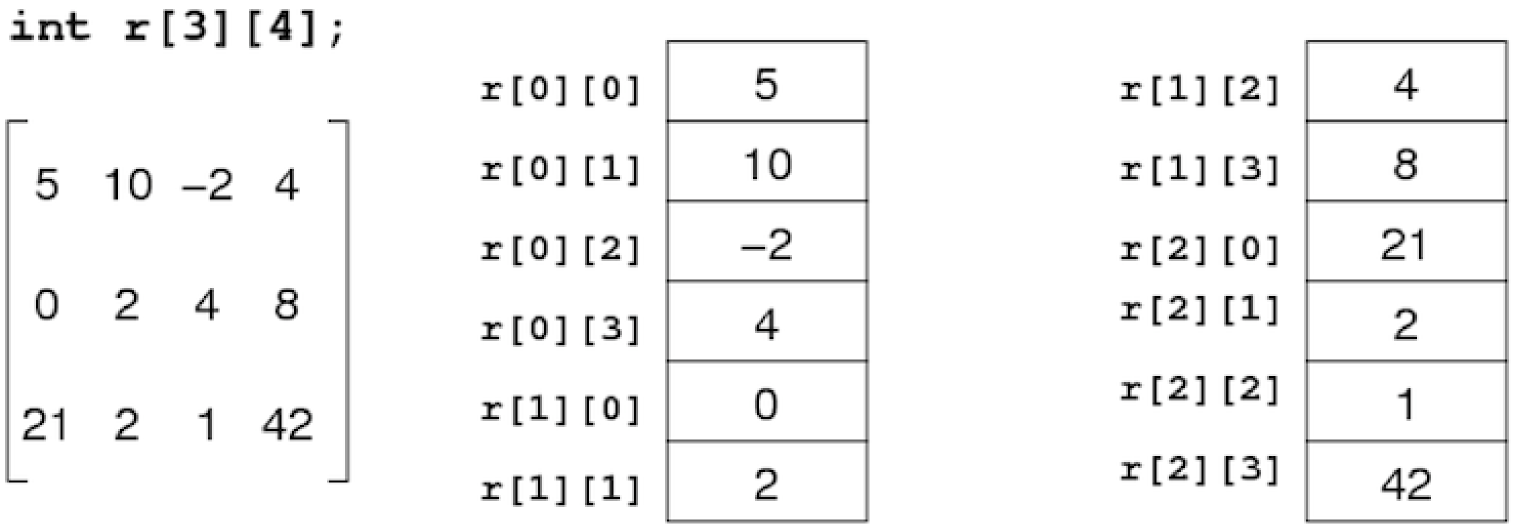
Note: **`q[0][1]==2.7`** **`r[1][3]==8`** **`q[1]=={3.1,0.1}`**

Multi-dimensional arrays can also be initialised:

```
float q[][] = {
   { 0.5, 2.7 },
   { 3.1, 0.1 }
};
```

Multi-dimensional Arrays (cont)

Storage representation of multi-dimensional arrays:



Multi-dimensional Arrays (cont)

Iteration can be done row-by-row or column-by-column:

```
int m[NROWS][NCOLS];
int row, col;

//row-by-row
for (row = 0; row < NROWS; row++) {
     for (col = 0; col < NCOLS; col++) {
          … m[row][col] …
     }
}
// colum-by-column
for (col = 0; col < NCOLS; col++) {
     for (row = 0; row < NROWS; row++) {
          … m[row][col] …
     }
}
```

Row-by-row is the most common style of iteration.

Defining New Data Types

C allows us to define new data type (names) via **`typedef`**:

```
typedef ExistingDataType NewTypeName;
```

Examples:

```
typedef float Temperature;

typedef int Matrix[20][20];
```

We will frequently use **`Bool`** whenever we want to stress the fact that we are interested in the logical rather than the numeric value of an expression:

```
typedef int Bool;
```

Defining New Data Types (cont)

Reasons to use **`typedef`**:

- give meaningful names to value types (documentation)
  - is a given number **`Temperature`**, **`Dollars`**, **`Volts`**, …?
- allow for easy changes to underlying type

```
typedef float Real;
Real complex_calculation(Real a, Real b) {
 Real c = log(a+b); … return c;
}
```

- "package up" complex type definitions for easy re-use
  - many examples to follow; **`Matrix`** is a simple example

Structures

A structure

- is a collection of variables, perhaps of different types, grouped together under a single name
- helps to organise complicated data into manageable entities
- exposes the connection between data within an entity
- is defined using the **`struct`** keyword

Example:

```
struct date {
        int day;
        int month;
        int year;
}; // don't forget the semicolon!
```

Structures (cont)

Defining a structure itself does not allocate any memory

We need to declare a variable in order to allocate memory

```
struct date christmas;
```

The components of the structure can be accessed using the "dot" operator

```
christmas.day   =   25;
christmas.month =   12;
christmas.year  = 2015;
```

Structures (cont)

A structure can be passed as a parameter to a function:

```
void print_date(struct date d) {

 printf("%d-%d-%d\n", d.day, d.month, d.year);
}

int is_leap_year(struct date d) {

 return ( ((d.year%4 == 0) && (d.year%100 != 0))
          || (d.year%400 == 0) );
}
```

Structures (cont)

One structure can be nested inside another:

```
struct date { int day, month, year; };

struct time { int hour, minute; };

struct speeding {
 char         plate[7];
 double       speed;
 struct date d;
 struct time t;
};
```

Structures (cont)

Possible memory layout produced for **`TicketT`** object:

```
---------------------------------
| D | S | A | 4 | 2 | X | \0|   |            7 bytes + 1 padding
---------------------------------
|                          68.4 |                          8 bytes
-------------------------------------------------
|            27 |             7 |          2017|    12 bytes
-------------------------------------------------
|            20 |            45 |                          8 bytes
---------------------------------
```

Note: padding is needed to ensure that **`plate`** lies on a 4-byte boundary.

Don't normally care about internal layout, since fields are accessed by name.

typedef and struct

We can also define a structured data type **TicketT** for speeding ticket objects:

```
typedef struct {
 int day, month, year;
} DateT;

typedef struct {
 int hour, minute;
} TimeT;

typedef struct {
 char   plate[7];    // e.g. "DSA42X"
 double speed;
        DateT  d;
 TimeT  t;
} TicketT;
```

typedef and struct (cont)

Note: structures can be defined in two different styles:

```
struct date { int day, month, year; };
// which would be used as
struct date somedate;

// or

typedef struct { int day, month, year; } DateT;
// which would be used as
DateT anotherdate;
```

The definitions produce objects with identical structures.

It is possible to combine both using the same identifier

```
typedef struct DateT { int day, month, year; } DateT;
// which could be used as
DateT        date1;
struct DateT date2;
```

typedef and struct (cont)

With the above **TicketT** type, we declare and use variables as …

```
#define NUM_TICKETS 1500

typedef struct {…} TicketT;

TicketT tickets[NUM_TICKETS];   // array of structs

// Print all speeding tickets in a readable format
for (i = 0; i < NUM_TICKETS; i++) {
    printf("%s %6.3f %d-%d-%d at %d:%d\n", tickets[i].plate,
        tickets[i].speed,
        tickets[i].d.day,
        tickets[i].d.month,
        tickets[i].d.year,
        tickets[i].t.hour,
        tickets[i].t.minute);
}
```

Summary

- Introduction to Algorithms and Data Structures
- C programming language, compiling with **`gcc`**
  - Basic data types (**`char`**, **`int`**, **`float`**)
  - Basic programming constructs (**`if`** … **`else`** conditionals, **`while`** loops, **`for`** loops)
  - Basic data structures (atomic data types, arrays, structures)

- Suggested reading (Moffat):
  - introduction to C … Ch.1; Ch.2.1-2.3, 2.5-2.6;
  - conditionals and loops … Ch.3.1-3.3; Ch.4.1-4.4
  - arrays … Ch.7.1,7.5-7.6
  - structures … Ch.8.1