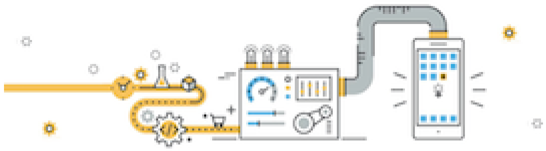
Week 1: Introduction - Elementary Data and Control Structures in C

COMP9024 20T2

1/104

Data Structures and Algorithms



Michael Thielscher

Web Site: www.cse.unsw.edu.au/~cs9024

Course Convenor

2/104

Name:	Michael Thielscher
Office:	K17-401J (turn left from lift and dial 57129)
Phone:	9385 7129
Email:	mit@unsw.edu.au
Consults:	personal: Thu 3-4pm, Email technical (course contents): Forum
Research:	Artificial Intelligence, Robotics, General Problem-Solving Systems
Pastimes:	Fiction, Films, Food, Football

... Course Convenor

3/104

Tutors:	Sahil Punchhi, s.punchhi@unsw.edu.au
	Kevin Luong, kevin.luong@unsw.edu.au
Course admin:	Michael Schofield, mschofield@cse.unsw.edu.au

Course Goals

4/104

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

Data structures

- how to store data inside a computer for efficient use

Algorithms

- step-by-step process for solving a problem (within finite amount of space and time)

... Course Goals

5/104

COMP9021 …

... Course Goals

6/104

COMP9024 …

Pre-conditions

7/104

There are no prerequisites for this course.
However we will move at fast pace through the necessary programming fundamentals. You may find it helpful if you are 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)
- use fundamental control structures (`if`, `while`, `for`)
- know fundamental programming techniques (recursion)
- 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)
  (graphs, search trees, …)
- choose/develop algorithms (A) on these DS
  (graph algorithms, tree algorithms, string algorithms, …)
- analyse performance characteristics of algorithms
- package a set of DS+A as an abstract data type
- develop and maintain C programs

Access to Course Material

All course information is placed on the main course website:

- www.cse.unsw.edu.au/~cs9024

Need to login to access material, submit homework and assignment, post on the forum, view your marks

Access livestream lectures, lecture recordings and quizzes (weeks 2, 4, 7, 9) on Moodle:

- COMP9024 Data Structures & Algorithms (T2-2020)

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, Goodrich and Tamassia's Java book, Skiena and Revilla's programming challenges book

---

Schedule

Week	**Lectures**	Assessment	Notes
1	Introduction, C language	—	
2	Analysis of algorithms	quiz	
3	Dynamic data structures	program	
4	Graph data structures	quiz	
5	Graph algorithms	program	
6	**Mid-term test (Thursday)**		Large Assignment
7	Search tree data structures	quiz	|
8	Search tree algorithms	program	|

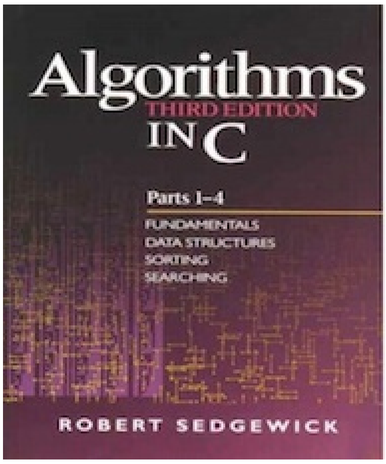
9	String algorithms, Approximation	quiz	|
10	Randomised algorithms, Review	program	| due

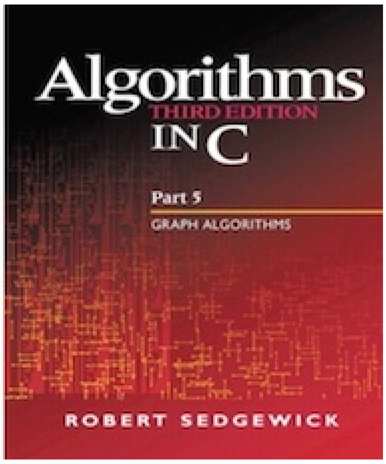
Resources

11/104

Textbook is a "double-header"

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





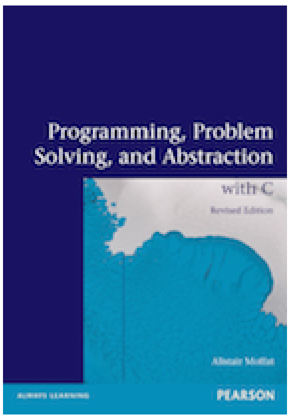
Good books, useful beyond COMP9024 (but coding style …)

... Resources

12/104

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

13/104

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!

Weekly Assessments

In weeks *(1), 3, 5, 8, 10* :

- you will be asked to submit 1 or 2 (small) programs
- which will be auto-marked against one or more test cases

In weeks *2, 4, 6, 8* :

- you will be given a short quiz (4-5 questions)
- with questions related to the exercises and the lecture

Programs and quizzes contribute 8% + 8% to overall mark.

- First assessment (week 1) is programming practice and will not count
  - Deadline: Tuesday, 9 June, 11:00:00am

Large Assignment

The large assignment gives you experience applying tools/techniques
(but to a larger programming problem than the homework)

The assignment will be carried out individually.

The assignment will be released after the mid-term test and is due in week 10.

The assignment contributes 12% to overall mark.

16.67% penalty will be applied to the maximum mark for every 24 hours late after the deadline.

- 1 day late: mark is capped at 10 (83.33% of the maximum possible mark)
- 2 days late: mark is capped at 8 (66.67% of the maximum possible mark)

- 3 days late: mark is capped at 6 (50% of the maximum possible mark)
- …

... Large Assignment

17/104

Advice on doing 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

18/104

Just Don't Do it

We get very annoyed by people who plagiarise.

... Plagiarism

19/104

Examples of Plagiarism (student.unsw.edu.au/plagiarism):

1. Copying

   Using same or similar idea *without acknowledging the source*
   This includes copying ideas from a website, internet

2. Collusion

   Presenting work as independent when produced in collusion with others
   This includes *students providing their work to another student*

   - which includes using any form of *publicly readable code repository*

Plagiarism will be checked for and punished (0 marks for assignment and, in severe cases/repeat offenders, 0 marks for course)

For COMP9024 you will need to complete a short new online course on Academic Integrity in Programming Courses

- We will ask for your completion certificate

Mid-term Test

20/104

1-hour online test in week 6 (*Thursday, 9 July, at time of the lecture*).

Format:

- some multiple-choice questions
- some descriptive/analytical questions with open answers

The mid-term test contributes 12% to overall mark.

Final Exam

21/104

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

Format:

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

The final exam contributes 60% to overall mark.

Must score at least 25/60 in the final exam to pass the course.

... Final Exam

22/104

How to pass the mid-term test and the Final Exam:

- do the Homework *yourself*
- do the Homework *every week*
- use C Practice Week 2 to practise programming in C
- practise programming outside classes
- read the lecture notes
- read the corresponding chapters in the textbooks

Assessment Summary

23/104

```
assn1 = mark for programs/quizzes (out of 8+8)
assn2 = mark for mid-term test    (out of 12)
assn3 = mark for large assignment (out of 12)
exam  = mark for final exam       (out of 60)

if (exam >= 25)
   total = assn1 + assn2 + assn3 + exam
else
   total = exam * (100/60)
```

To pass the course, you must achieve:

- at least 25/60 for `exam`
- at least 50/100 for `total`

Summary

24/104

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 software design and development process

C Programming Language

Why C?

26/104

- good example of an imperative language
- gives the programmer great control
- produces fast code
- many libraries and resources
- main language for writing operating systems and compilers; and commonly used for a variety of applications in industry (and science)

Brief History of C

27/104

C and UNIX operating system share a complex history …

- C was originally designed for and implemented on UNIX
- 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
- American National Standards Institute (ANSI) C standard published in 1988
    - this greatly improved source code portability
- Current standard: C11 (published in 2011)

Basic Structure of a C Program

28/104

```
// 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?

29/104

```
#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

30/104

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 and 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

31/104

```
#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`

32/104

C source code: prog.c
↓
a.out **(executable program)**

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`

33/104

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**

Sidetrack: Printing Variable Values with `printf()`

34/104

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

printf(*format-string*, *expr*$_1$, *expr*$_2$, …);

format-string can use the following placeholders:

%d	decimal	%f	floating-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.

id  = 'z';
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
  ```

Algorithms in C

Basic Elements

36/104

Algorithms are built using

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

Assignments

37/104

- In C, each statement is terminated by a semicolon **;**
- Curly brackets **{ }** used to enclose statements in a block
- Usual arithmetic operators: **+, -, *, /, %**
- Usual assignment operators: **=, +=, -=, *=, /=, %=**
- 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

38/104

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();
}
```

is often written as

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

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

39/104

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

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

---

1. `a == 3, b == 3`
2. `a == 5, b == 4`

Conditionals

41/104

```
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 `{ }`

42/104

... Conditionals

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 (my preference)
if (x) {
   statements;
}
```

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

... Conditionals

43/104

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

44/104

1. What is the output of the following program fragment?

```
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`

Loops

46/104

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

47/104

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?

48/104

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

```
88
87
..
81

44
..
41

22
```

```
21
```

Functions

50/104

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

51/104

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

52/104

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:

```
// Euclid's gcd algorithm (recursive version)
int euclid_gcd(int m, int n) {
    if (n == 0) {
        return m;
    } else {
        return euclid_gcd(n, m % n);
    }
}
```

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

```
return;
```

Data Structures in C

Basic Data Types

54/104

- 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;
```

Aggregate Data Types

55/104

Families of aggregate data types:

- homogeneous … all elements have same base type
  - arrays (e.g. `char s[50]`,`int v[100]`)
- heterogeneous … elements may combine different base types
  - structures (e.g. `struct student { char name[30]; int zID; }`)

Arrays

56/104

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

57/104

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];
}
```

Sidetrack: C Style

58/104

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

```
#define SPEED_OF_LIGHT 299792458.0
#define ERROR_MESSAGE "Out of memory.\n"
```

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

---

... Sidetrack: C Style

59/104

UNSW Computing provides a style guide for C programs:

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

Not strictly mandatory for COMP9024, but very useful guideline

Style considerations that *do* matter for your COMP9024 assignments:

- use proper layout, including consistent indentation
  - 3 spaces throughout, or 4 spaces throughout
    do *not* use TABs
- keep functions short and break into sub-functions as required
- use meaningful names (for variables, functions etc)
- use symbolic constants to avoid burying "magic numbers" in the code
- comment your code

... Sidetrack: C Style

60/104

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: C Style

61/104

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: C Style

62/104

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 _____;}
```

Strings

63/104

"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

64/104

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? 65/104

```
#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"
```

Sidetrack: Reading Variable Values with `scanf()` and `atoi()` 67/104

Formatted input read from standard input (e.g. keyboard)

`scanf(`*format-string*, *expr*$_1$, *expr*$_2$, *…*`);`

Converting string into integer

`int value = atoi(`*string*`);`

Example:

```
#include <stdio.h>   // includes definition of BUFSIZ (usually =512) and scanf()
#include <stdlib.h>  // includes definition of atoi()

...

char str[BUFSIZ];
```

```
int n;

printf("Enter a string: ");
scanf("%s", str);
n = atoi(str);
printf("You entered: \"%s\". This converts to integer %d.\n", str, n);
```

```
Enter a string: 9024
You entered: "9024". This converts to integer 9024.
```

Arrays and Functions

68/104

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

69/104

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

70/104

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 72/104

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 }
};
```

Sidetrack: Defining New Data Types 73/104

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

`typedef` *`ExistingDataType`* *`NewTypeName`*`;`

Examples:

```
typedef float Temperature;
```

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

... Sidetrack: Defining New Data Types 74/104

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:

```
typedef struct {
       char name[30];
       int  zID;
} StudentT;
```

... Structures

One structure can be *nested* inside another:

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

typedef struct {
        int hour, minute;
} TimeT;

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

... Structures

Possible memory layout produced for `TicketT` object:

```
---------------------------------
| D | S | A | 4 | 2 | X | \0|   |    7 bytes + 1 padding
---------------------------------
|                          68.4 |    8 bytes
---------------------------------
|             2 |             6 |    8 bytes
---------------------------------
|            20 |            45 |    8 bytes
---------------------------------
```

Note: padding is needed to ensure that `plate` lies on an 8-byte block.

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

... Structures

78/104

Defining a structured data type itself does not allocate any memory

We need to declare a variable in order to allocate memory

```
DateT christmas;
```

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

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

... Structures

79/104

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 %6.2f %d/%d at %d:%d\n", tickets[i].plate,
                                        tickets[i].speed,
                                        tickets[i].d.day,
                                        tickets[i].d.month,
                                        tickets[i].t.hour,
                                        tickets[i].t.minute);
}

// Sample output:
//
// DSA42X  68.40 2/6 at 20:45
```

... Structures

80/104

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

```
void print_date(DateT d) {

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

int is_winter(DateT d) {

        return ( (d.month >= 6) && (d.month <= 8) );
}
```

Data Abstraction

Abstract Data Types

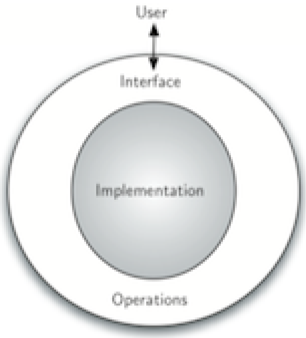
82/104

A *data type* is …

- a set of *values* (atomic or structured values) e.g. *integer stacks*
- a collection of *operations* on those values e.g. *push, pop, isEmpty?*

An *abstract data type* …

- is a logical description of how we view the data and operations
- without regard to how they will be implemented
- creates an *encapsulation* around the data
- is a form of *information hiding*



... Abstract Data Types

83/104

Users of the ADT see only the *interface*

Builders of the ADT provide an *implementation*

ADT *interface* provides

- a user-view of the data structure
- function signatures (prototypes) for all operations
- semantics of operations (via documentation)
- ⇒ a "contract" between ADT and its clients

ADT *implementation* gives

- concrete definition of the data structures
- function implementations for all operations

... Abstract Data Types

84/104

ADT interfaces are *opaque*

- clients *cannot* see the implementation via the interface

ADTs are important because …

- facilitate decomposition of complex programs
- make implementation changes invisible to clients
- improve readability and structuring of software
- allow for reuse of modules in other systems

ADOs and ADTs

85/104

We want to distinguish …

- ADO = *a*bstract *d*ata *o*bject
- ADT = *a*bstract *d*ata *t*ype

Warning: Sedgewick's first few examples are ADOs, not ADTs.

Example: Abstract Stack Data Object

86/104

Stack, aka *pushdown stack* or *LIFO data structure* (last in, first out)

Assume (for the time being) stacks of `char` values

Operations:

- *create* an empty stack
- insert (*push*) an item onto stack
- remove (*pop*) most recently pushed item
- check whether stack *is empty*

Applications:

- undo sequence in a text editor
- bracket matching algorithm
- …

... Example: Abstract Stack Data Object

87/104

Example of use:

Stack	Operation	Return value
?	create	-
-	isempty	true
-	push a	-
a	push b	-
a b	push c	-
a b c	pop	c
a b	isempty	false

Stack vs Queue

88/104

Queue, aka *FIFO data structure* (first in, first out)

Insert and delete are called *enqueue* and *dequeue*

Applications:

- the checkout at a supermarket
- people queueing to go onto a bus

- objects flowing through a pipe (where they cannot overtake each other)
- chat messages
- web page requests arriving at a web server
- printing jobs arriving at a printer
- …

Exercise #7: Stack vs Queue

89/104

Consider the previous example but with a queue instead of a stack.

Which element would have been taken out ("dequeued") first?

---

a

Stack as ADO

91/104

Interface (a file named `Stack.h`)

```
// Stack ADO header file

#define MAXITEMS 10

void StackInit();       // set up empty stack
int  StackIsEmpty();    // check whether stack is empty
void StackPush(char);   // insert char on top of stack
char StackPop();        // remove char from top of stack
```

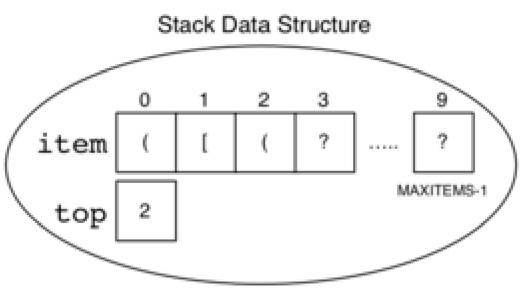
Note:

- no explicit reference to Stack object
- this makes it an *Abstract Data Object (ADO)*

... Stack as ADO

92/104

Implementation may use the following data structure:



... Stack as ADO

93/104

Implementation (in a file named `Stack.c`):

```
#include "Stack.h"
```

```
#include <assert.h>

// define the Data Structure
typedef struct {
   char item[MAXITEMS];
   int  top;
} stackRep;

// define the Data Object
static stackRep stackObject;

// set up empty stack
void StackInit() {
   stackObject.top = -1;
}

// check whether stack is empty
int StackIsEmpty() {
   return (stackObject.top < 0);
}
```

```
// insert char on top of stack
void StackPush(char ch) {
   assert(stackObject.top < MAXITEMS-1);
   stackObject.top++;
   int i = stackObject.top;
   stackObject.item[i] = ch;
}

// remove char from top of stack
char StackPop() {
   assert(stackObject.top > -1);
   int i = stackObject.top;
   char ch = stackObject.item[i];
   stackObject.top--;
   return ch;
}
```

- **`assert(`*`test`*`)`** terminates program with error message if *test* fails
- `static` *`Type Var`* declares *`Var`* as *local* to `Stack.c`

Exercise #8: Bracket Matching

94/104

Bracket matching … check whether all opening brackets such as '(', '[', '{' have matching closing brackets ')', ']', '}'

Which of the following expressions are balanced?

1. `(a+b) * c`
2. `a[i]+b[j]*c[k])`
3. `(a[i]+b[j])*c[k]`
4. `a(a+b]*c`
5. `void f(char a[], int n) {int i; for(i=0;i<n;i++) { a[i] = (a[i]*a[i])*(i+1); }}`
6. `a(a+b * c`

---

1. balanced
2. not balanced (case 1: an opening bracket is missing)
3. balanced
4. not balanced (case 2: closing bracket doesn't match opening bracket)
5. balanced
6. not balanced (case 3: missing closing bracket)

... Stack as ADO

96/104

Bracket matching algorithm, to be implemented as a *client* for Stack ADO:

```
bracketMatching(s):
|  Input  stream s of characters
|  Output true if parentheses in s balanced, false otherwise
|
|  for each ch in s do
|  |  if ch = open bracket then
|  |     push ch onto stack
|  |  else if ch = closing bracket then
|  |  |  if stack is empty then
|  |  |     return false                    // opening bracket missing (case 1)
|  |  |  else
```

```
|  |  |      pop top of stack
|  |  |      if brackets do not match then
|  |  |         return false               // wrong closing bracket (case 2)
|  |  |      end if
|  |  |   end if
|  |  end if
|  end for
|  if stack is not empty then return false  // some brackets unmatched (case 3)
|                         else return true
```

... Stack as ADO

97/104

Execution trace of client on sample input:

```
( [ { } ] )
```

Next char	Stack	Check
-	empty	-
(	(	-
[	([	-
{	([{	-
}	([	{ vs } ✓
]	(	[vs] ✓
)	empty	(vs) ✓
eof	empty	-

Exercise #9: Bracket Matching Algorithm

98/104

Trace the algorithm on the input

```
void f(char a[], int n) {
   int i;
   for(i=0;i<n;i++) { a[i] = a[i]*a[i])*(i+1); }
}
```

Next bracket	Stack	Check
start	empty	-
(	(	-
[	([	-
]	(	✓
)	empty	✓
{	{	-
(	{ (	-
)	{	✓

{	{ {	-
[	{ { [	-
]	{ {	✓
[	{ { [	-
]	{ {	✓
[	{ { [	-
]	{ {	✓
)	{	false

Exercise #10: Implement Bracket Matching Algorithm in C

100/104

- Use Stack ADT

```
#include "Stack.h"
```

- *Sidetrack: Character I/O Functions in C* (requires `<stdio.h>`)

```
int getchar(void);
```

  - returns character read from standard input as an **`int`**, or returns **`EOF`** on end of file (keyboard: CTRL-D on Unix, CTRL-Z on Windows)

```
int putchar(int ch);
```

  - writes the character `ch` to standard output
  - returns the character written, or `EOF` on error

Managing Abstract Data Types and Objects in C

Compilers

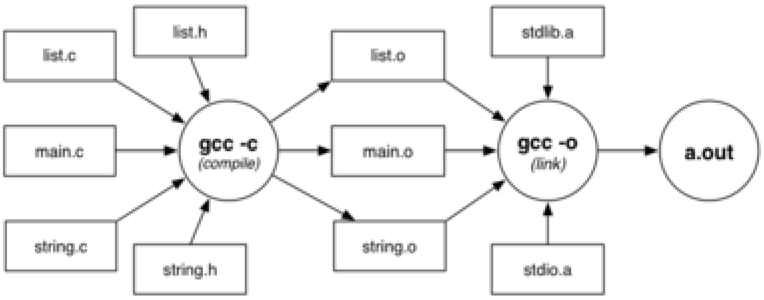
102/104

Compilers are programs that

- convert program source code to executable form
- "executable" might be machine code or bytecode

The Gnu C compiler (**`gcc`**)

- applies source-to-source transformation (pre-processor)
- compiles *source code* to produce *object files*
- links object files and *libraries* to produce *executables*



... Compilers

103/104

Compilation/linking with `gcc`

```
gcc -c Stack.c
produces Stack.o, from Stack.c and Stack.h

gcc -c brackets.c
produces brackets.o, from brackets.c and Stack.h

gcc -o rbt brackets.o Stack.o
links brackets.o, Stack.o and libraries
producing executable program called rbt
```

Note that `stdio,assert` included implicitly.

`gcc` is a multi-purpose tool

- compiles (`-c`), links, makes executables (`-o`)

Summary

104/104

- 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)
- Introduction to ADTs
  - Compilation

- 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
- Suggested reading (Sedgewick):
  - introduction to ADTs … Ch. 4.1-4.3

Produced: 29 May 2020