Assignment 3

General Info

You must read fully and carefully the assignment specification and instructions.

- **Course:** COMP20003 Algorithms and Data Structures @ Semester 2, 2022
- **Deadline Submission:** Friday 21th October 2022 @ Midday (almost end of Week 12)
- **Course Weight:** 15%
- **Assignment type:** individual
- **ILOs covered:** 2,4
- **Submission method:** via ED

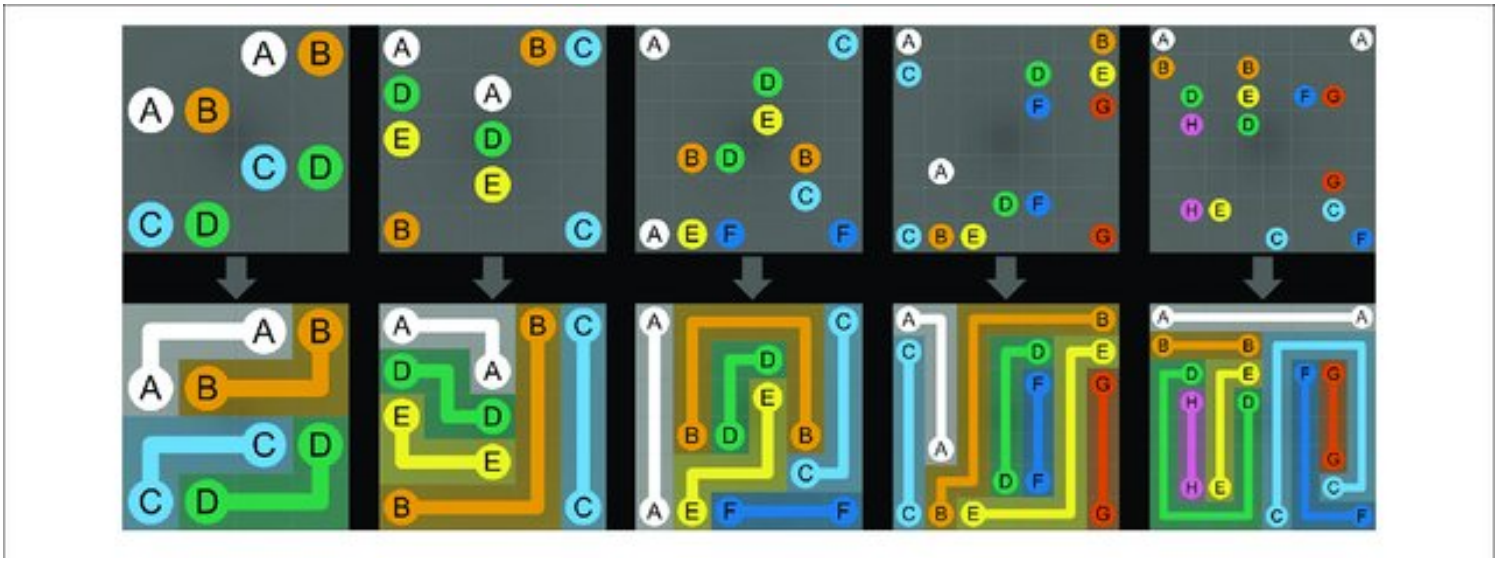
Purpose

The purpose of this assignment is for you to:

- Increase your proficiency in C programming, your dexterity with dynamic memory allocation and your understanding of data structures, through programming a search algorithm over Graphs.
- Gain experience with applications of graphs and graph algorithms to solving combinatorial games, one form of artificial intelligence.

Walkthrough

Flow Free - a circuit design problem

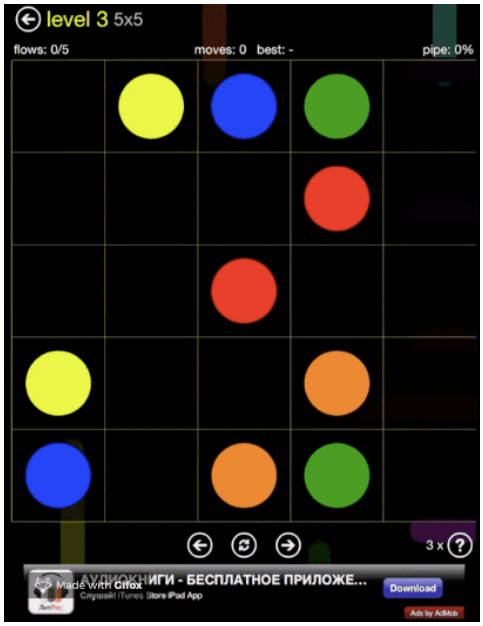


In this programming assignment you'll be expected to build a solver for the Flow Free game. Its origins can be traced back to 1897, a puzzle published by the professional puzzler Sam Loyd, in a column he wrote for the *Brooklyn Daily Eagle (more details in the Matematica Journal).* This puzzle was popularised in Japan under the name of Numberlink.

You can play the game for free on your android/apple phone/pc, over a web browser or solve it compiling the code given to you.

The code in this assignment was adapted from the open-source terminal version made available by mzucker.

Game Rules



The game presents a **grid** with colored dots occupying some of the squares. The objective is to connect dots of the same color by drawing *pipes* between them such that the entire grid is occupied by pipes. However, pipes may not intersect.

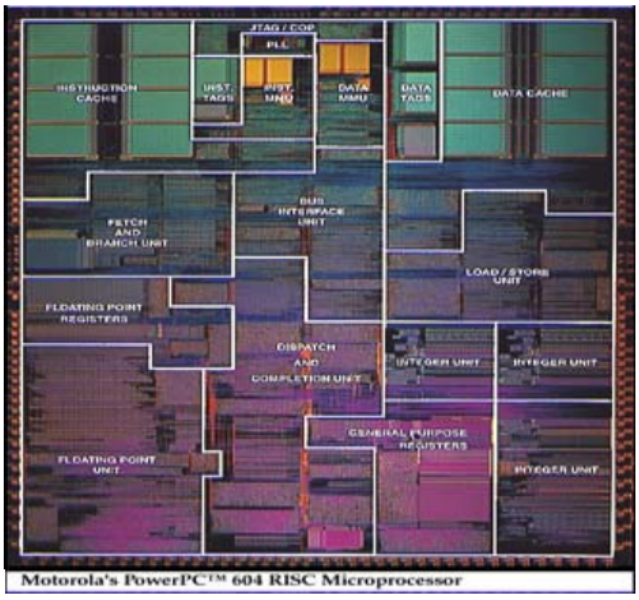
The difficulty is determined by the size of the grid, ranging from 5x5 to 15x15 squares.

For the courious reader

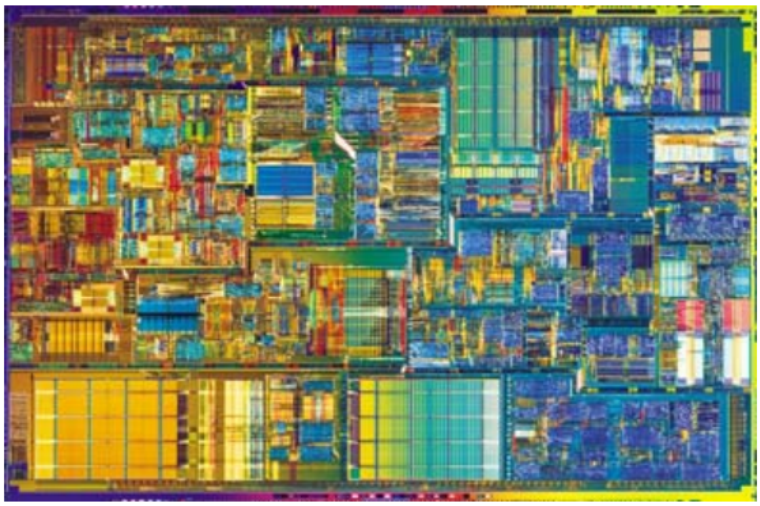
The Science of Flow Free

Flow Free can be studied using the theory of computational complexity. The game is yet another proof that William Rowan Hamilton was ahead of his time: not only Candy Crash is addictive and NP-complete, *Flow Free* is another instance of a NP-complete game being wildly successful and addictive. Here's a list of other famous arcade games that are known to be NP-complete.

I consider games such as Flow Free in the category of *serious games,* as solving them efficiently can have a huge impact on a wide range of routing problems. For Example, Flow Free problem is similar to the problem of circuit placement and interconnection design, nowadays known as one of the key problems in Very Large Scale Integration (VLSI) design, i.e. microchip design. If interested, you can read the intro (specifically section 1.3.1) of this book and section D of this paper: An industrial view of electronic design automation.



PowerPC 604



Pentium 4

The figure above shows some serious Flow Free designs. More layouts can be found in these slides. Chips look like abstract modern art ;)

NP-complete problems are hard to solve. The best algorithms to solve NP-Complete problems run in exponential time as a function of the size of the problem and the shortest accepted solution. Hence,

be aware that your Flow Free/AI solver will struggle more as the size of the grid increases. We talked in the lectures about the Travelling Salesman Problem as one example of an NP-Complete problem. In fact, many real-world problems fall under this category.

Flow Free is difficult not only because of its large branching factor, but also because of its large search tree depth. Skilled human players rely mostly on heuristics and are usually able to quickly discard a great many futile or redundant lines of play by recognizing patterns, dead-ends, and subgoals, thereby drastically reducing the amount of search.

Flow Free puzzles can be solved automatically by using a classic single-agent search AI algorithm, such as A*; enhanced by several techniques that make use of domain-specific knowledge.[8]

The Algorithm

Each possible configuration of the Flow Free grid is called a *state*. Each position in the grid can be free or contain a color. The Flow Free Graph $G = \langle V, E \rangle$ is implicitly defined, i.e. the graph is not completely loaded in memory at once, instead, it starts only with the initial state, and the remainder of the graph is discovered while searching for a solution. The vertex set V is defined as all the possible configurations (states), and the edges E connecting two vertexes are defined by the legal movements **(right, left, up, down) for each color.** The code provided makes sure that only legal moves are generated:

> A move is legal only if it extends a color pipe. A color cannot be placed if it's not adjacent to the pipe of the same color. Furthermore, to decrease the number of edges in the Graph, the pipe can only be started from one of the two initial colors, making one of them the start and the other the goal of the pipe.

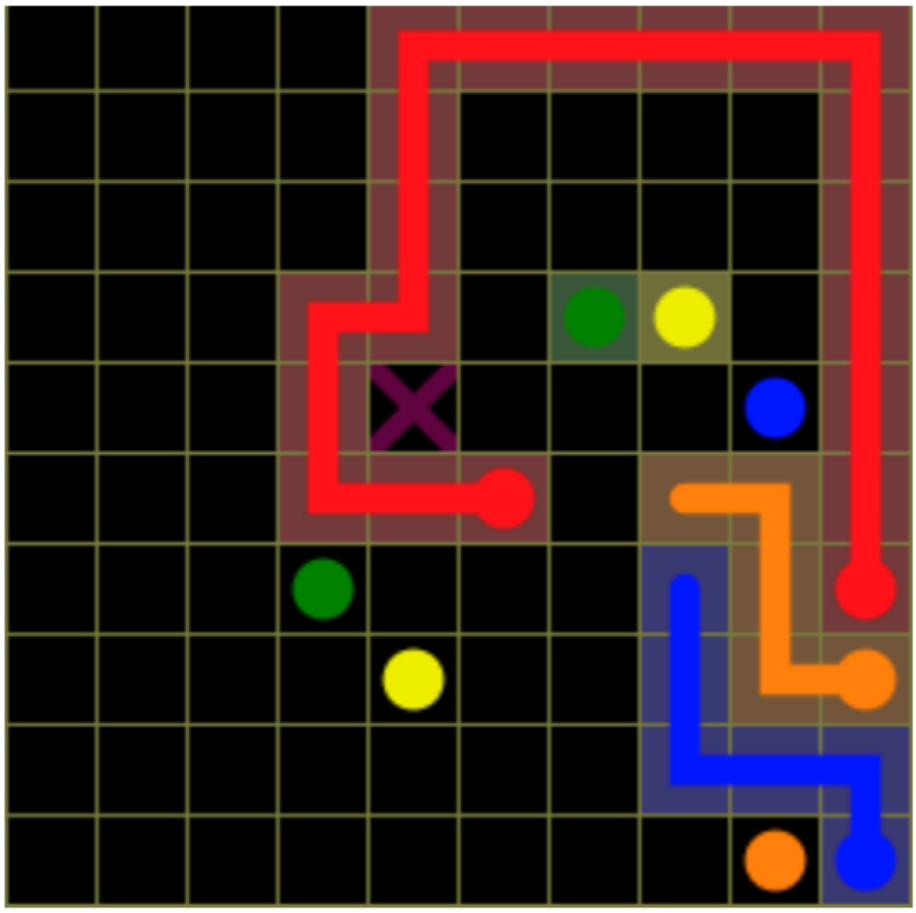
All edges have a weight of 1.

Your task is to find the path traversing the Flow Free Graph from the initial state (vertex) leading to a state (vertex) where all the grid cells contain a color, and therefore finding a solution to the puzzle. A path is a sequence of movements painting a new cell in the grid. You are going to use Dijkstra to find the shortest path first, along with some game-specific optimizations to speed up your algorithm.

The initial node (vertex) corresponds to the initial configuration (Algorithm 1, line 1). The algorithm selects a node to expand (generate the possible pipe extensions, line 5) along with an ordering of which is the *next color* the algorithm should try to connect (line 6). Therefore, when a node is expanded, it has at most 4 children (right, left, up, down). Once all the children have been added to the priority queue, the algorithm will pick again the next node to extend (line 5), as well as the *next color to consider,* until a solution is found. Different color orderings can be tried with the command line options (type to see all options). Each ordering has a different impact on the number of nodes needed by Dijkstra to find a solution. By default, the search will expand a maximum of 1GB of nodes (line 12). This is checked by a variable named .

```
GRAPHSEARCH(Graph, start)
 1  node ← start
 2  pq ← priority Queue Containing start node Only
 3  while frontier ≠ empty
 4  do
 5      node ← pq.pop()
 6      nextColor ← select next color in ordering
 7      for each move direction
 8      do
 9          if move direction for nextColor is valid
10            then
11                  newNode ← applyMoveDirection(node)
12                  if number of nodes in pq ≥ max_nodes
13                    then
14                          result = SEARCH_FULL
15                          break
16
17                  if newNode is a dead end
18                    then
19                          deleteNewNode
20                          continue
21
22                  if newNode is thesolution
23                    then
24                          solution = NewNode
25                          result = SEARCH_SUCCESS
26                          break
27
28                  pq.push(newNode)
29
30
31  free priority queue
32  return result
```

When you you have to create a new node that points to the parent, updates the grid resulting from applying the direction chosen, updates the priority of the node, and updates any other auxiliary data in the node. The priority of the node is given by the length of the path from the root node, i.e. how many grid cells have been painted. Check the file as this function is already given.



A *dead-end* is a configuration for which you know a solution cannot exist. The figure above shows an example of a dead-end: the cell marked with X cannot be filled with any color. The problem is unsolvable with the current configuration, and we do not need to continue searching any of its successors. You can recognize dead-end cells by looking for a **free cell** in the grid that is **surrounded by three completed path segments (colored) or walls**. The current position of an in-progress pipe and goal position have to be treated as free space. Detecting unsolvable states early can prevent lots of unnecessary search. The search will eventually find a solution. If we recognize dead-ends and do not generate their successors (pruning), the search for a solution will likely take less time and memory, as you'll avoid exploring all possible successors of an unsolvable state.

You might have multiple paths leading to a solution. **Your algorithm should consider the possible actions in the following order**: left (0), right (1), up (2) or down (3).

Make sure you manage the memory well. When you finish running the algorithm, you have to free all the nodes from the memory, otherwise you will have memory leaks. You will notice that the algorithm can run out of memory fairly fast after expanding millions of nodes.

Check the files and where you'll find many of the functions in the algorithm already implemented. Other useful functions are located directly in the file , which is the only file you need to edit to write your algorithm inside the function . For dead-end detection, you need to look at , function . Look for the comment . All the files are in the folder .

Deliverables

Deliverable 1 - Dijkstra **Solver** source code

You are expected to hand in the source code for your solver, written in C. Obviously, your source code is expected to compile and execute flawlessly using the following makefile command:

generating an executable called

Remember to compile using the optimization flag for doing your experiments, it will run twice as quickly as compiling with the debugging flag (see). The provided Makefile compiles with the optimization flag by default, and with the debugging flag if you type . For the submission, please **do not remove the option from your Makefile**, as our scripts need this flag for testing. Your program must not be compiled under any flags that prevent it from working under or .

Your implementation should be able to solve the *regular* puzzles provided. To solve the *extreme* puzzles, you'll need further enhancements that go beyond the time for this assignment, but feel free to challenge yourself if you finish early and explore how you would solve the extreme puzzles.

Remember to implement both the **AI solver** and the **dead-end detection**.

Deliverable 2 – Experimentation

Besides handing in the solver source code, you're required to provide a table reporting at least the execution time and number of generated nodes with and without dead-end detection. Include in the table only the puzzles that your solver finds a solution to.

Plot figures, where the x-axis can be the number of free cells at the start, or the size of the grid, and the y-axis is either the number of generated states or solution time.

Explain your results using your figures and tables. Which complexity growth does your data show? What's the computational benefit of the dead-end detection, does it decrease the growth rate? Answer concisely.

If you decide to implement any further optimization beyond the instructions of the assignment, or change the default arguments such as *allowed memory* or *color ordering*, please discuss their impact on the experimentation section as well.

Please include your Username, Student ID and Full Name in your Document.

My recommendation is that you generate the plots using any standard Python visualization library. See for example Seaborn or Matplotlib. Otherwise, there's always the old-school excel/open-office/google-sheets method.

Deliverable 3 - Submission Certification Form

Once you submit your code, please also fill out the form, no later than 24h after the deadline.

The form is mandatory, and shouldn't take more than 5 minutes.

Rubric

Assignment marks will be divided into three different components.

1. Solver (8)
2. Dead-End Detection(2)
3. Experimentation (3)
4. Code Style (2)

Please note that you should be ready to answer any question we might have on the details of your assignment solution by e-mail, or even attending a brief interview with me, in order to clarify any doubts we might have.

The Code Base

You are given a base code. You can compile the code and execute the solver by typing . You are going to have to program your solver in the file . Look at the file and implement the missing part in the function called . Once you implement the search algorithm, go to the file called and implement the function called

You are given the structure of a node in files, and also a priority queue implementation. Look into the and files to know about the functions you can call to perform the search.

In your final submission, you are free to change any file, but make sure the command line options remain the same.

Input

In order to execute your solver use the following command:

```
./flow [options]  <puzzleName1> ... <puzzleNameN>
```

for example:

```
 ./flow puzzles/regular_5x5_01.txt
```

Will run the solver for the regular 5 by 5 puzzle, and report if the search was *successful*, the *number of nodes generated* and the *time* taken. if you use (quiet) it will report the solutions more concisely. This option can be useful if you want to run several puzzles at once and study their performance.

If you append the option it will *animate* the solution found. If you append the option it will use the *dead-end detection* mechanism that you implemented. Feel free to explore the impact of the other options, specifically the *ordering* in which the colors are explored. By default, the color that has fewer free neighbors (most constrained), is the one that is going to be considered first.

All the options can be found if you use option :

```
$./flow -h
usage: flow_solver [ OPTIONS ] BOARD1.txt
BOARD2.txt [ ... ] ]

Display options:

  -q, --quiet             Reduce output
```

```
  -d, --diagnostics       Print diagnostics when search unsuccessful
  -A, --animation         Animate solution
  -F, --fast              Speed up animation 4x
  -C, --color             Force use of ANSI color
  -S, --svg               Output final state to SVG

Node evaluation options:

  -d, --deadends          dead-end checking

Color ordering options:

  -r, --randomize         Shuffle order of colors before solving
  -c, --constrained       Disable order by most constrained

Search options:

  -n, --max-nodes N       Restrict storage to N nodes
  -m, --max-storage N     Restrict storage to N MB (default 1024)

Help:

  -h, --help              See this help text
```

Output

Your solver will print the following information if option is used:

1. Puzzle Name
2. SearchFlag (see , line 65-68 to undestand the flags)
3. Total Search Time, in seconds
4. Number of generated nodes
5. A final Summary

For example, the output of your solver could be:

```
../puzzles/regular_5x5_01.txt s 0.000 18
../puzzles/regular_6x6_01.txt s 0.000 283
../puzzles/regular_7x7_01.txt s 0.002 3,317
../puzzles/regular_8x8_01.txt s 0.284 409,726
../puzzles/regular_9x9_01.txt s 0.417 587,332
5 total s 0.704 1,000,676
```

These numbers depend on your implementation of the search, the ordering you use, and whether you prune dead-ends. If we use dead-end pruning we get the following results

```
../puzzles/regular_5x5_01.txt s 0.000 17
../puzzles/regular_6x6_01.txt s 0.000 254
```

```
../puzzles/regular_7x7_01.txt s 0.001 2,198
../puzzles/regular_8x8_01.txt s 0.137 182,136
../puzzles/regular_9x9_01.txt s 0.210 279,287
5 total s 0.349 463,892
```

Remember that in order to get full marks, your solver has to solve at least the regular puzzles.

Programming Style

Below is a style guide which assignments are evaluated against. For this subject, the 80 character limit is a guideline rather than a rule - if your code exceeds this limit, you should consider whether your code would be more readable if you instead rearranged it.

```
/** ***********************
* C Programming Style for Engineering Computation
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au) 13/03/2011
* Definitions and includes
* Definitions are in UPPER_CASE
* Includes go before definitions
* Space between includes, definitions and the main function.
* Use definitions for any constants in your program, do not just write them
* in.
*
* Tabs may be set to 4-spaces or 8-spaces, depending on your editor. The code
* Below is ``gnu'' style. If your editor has ``bsd'' it will follow the 8-space
* style. Both are very standard.
*/

/**
* GOOD:
*/

#include <stdio.h>
#include <stdlib.h>
#define MAX_STRING_SIZE 1000
#define DEBUG 0
int main(int argc, char **argv) {
 ...

/**
* BAD:
*/

/* Definitions and includes are mixed up */
#include <stdlib.h>
#define MAX_STING_SIZE 1000
/* Definitions are given names like variables */
#define debug 0
#include <stdio.h>
/* No spacing between includes, definitions and main function*/
int main(int argc, char **argv) {
 ...

/** ****************************
* Variables
* Give them useful lower_case names or camelCase. Either is fine,
```

```
* as long as you are consistent and apply always the same style.
* Initialise them to something that makes sense.
*/

/**
* GOOD: lower_case
*/

int main(int argc, char **argv) {

  int i = 0;
  int num_fifties = 0;
  int num_twenties = 0;
  int num_tens = 0;

 ...
/**
* GOOD: camelCase
*/

int main(int argc, char **argv) {

  int i = 0;
  int numFifties = 0;
  int numTwenties = 0;
  int numTens = 0;

 ...
/**
* BAD:
*/

int main(int argc, char **argv) {

  /* Variable not initialised - causes a bug because we didn't remember to
  * set it before the loop */
  int i;
  /* Variable in all caps - we'll get confused between this and constants
  */
  int NUM_FIFTIES = 0;
  /* Overly abbreviated variable names make things hard. */
  int nt = 0

  while (i < 10) {
    ...
    i++;
  }

 ...

/** ********************
* Spacing:
* Space intelligently, vertically to group blocks of code that are doing a
* specific operation, or to separate variable declarations from other code.
```

```
* One tab of indentation within either a function or a loop.
* Spaces after commas.
* Space between ) and {.
* No space between the ** and the argv in the definition of the main
* function.
* When declaring a pointer variable or argument, you may place the asterisk
* adjacent to either the type or to the variable name.
* Lines at most 80 characters long.
* Closing brace goes on its own line
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

 int i = 0;

 for(i = 100; i >= 0; i--) {
   if (i > 0) {
     printf("%d bottles of beer, take one down and pass it around,"
     " %d bottles of beer.\n", i, i - 1);
   } else {
     printf("%d bottles of beer, take one down and pass it around."
     " We're empty.\n", i);
   }
 }

 return 0;
}

/**
* BAD:
*/

/* No space after commas
* Space between the ** and argv in the main function definition
* No space between the ) and { at the start of a function */
int main(int argc,char ** argv){
 int i = 0;
 /* No space between variable declarations and the rest of the function.
 * No spaces around the boolean operators */
 for(i=100;i>=0;i--) {
 /* No indentation */
 if (i > 0) {
 /* Line too long */
 printf("%d bottles of beer, take one down and pass it around, %d
bottles of beer.\n", i, i - 1);
 } else {
 /* Spacing for no good reason. */

 printf("%d bottles of beer, take one down and pass it around."
 " We're empty.\n", i);
```

```
 }
 }
 /* Closing brace not on its own line */
 return 0;}

/** ****************
* Braces:
* Opening braces go on the same line as the loop or function name
* Closing braces go on their own line
* Closing braces go at the same indentation level as the thing they are
* closing
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

  ...

  for(...) {
    ...
  }

  return 0;
}

/**
* BAD:
*/

int main(int argc, char **argv) {

  ...

  /* Opening brace on a different line to the for loop open */
  for(...)
  {
    ...
    /* Closing brace at a different indentation to the thing it's
    closing
    */
           }

 /* Closing brace not on its own line. */
       return 0;}

/** **************
* Commenting:
* Each program should have a comment explaining what it does and who created
* it.
* Also comment how to run the program, including optional command line
```

```
* parameters.
* Any interesting code should have a comment to explain itself.
* We should not comment obvious things - write code that documents itself
*/

/**
* GOOD:
*/

/* change.c
*
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au)
13/03/2011
*
* Print the number of each coin that would be needed to make up some
change
* that is input by the user
*
* To run the program type:
* ./coins --num_coins 5 --shape_coins trapezoid --output blabla.txt
*
* To see all the input parameters, type:
* ./coins --help
* Options::
*   --help                Show help message
*   --num_coins arg       Input number of coins
*   --shape_coins arg     Input coins shape
*   --bound arg (=1)      Max bound on xxx, default value 1
*   --output arg          Output solution file
*
*/

int main(int argc, char **argv) {

  int input_change = 0;

  printf("Please input the value of the change (0-99 cents
  inclusive):\n");
  scanf("%d", &input_change);
  printf("\n");

  // Valid change values are 0-99 inclusive.
  if(input_change < 0 || input_change > 99) {
    printf("Input not in the range 0-99.\n")
  }

  ...

/**
* BAD:
*/

/* No explanation of what the program is doing */
int main(int argc, char **argv) {
```

```
  /* Commenting obvious things */
  /* Create a int variable called input_change to store the input from
  the
  * user. */
  int input_change;

  ...

/** ****************
* Code structure:
* Fail fast - input checks should happen first, then do the computation.
* Structure the code so that all error handling happens in an easy to read
* location
*/

/**
* GOOD:
*/
if (input_is_bad) {
  printf("Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}

/* Do computations here */
...

/**
* BAD:
*/

if (input_is_good) {
  /* lots of computation here, pushing the else part off the screen.
  */
  ...
} else {
  fprintf(stderr, "Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}
```

Some automatic evaluations of your code style may be performed where they are reliable. As determining whether these style-related issues are occurring sometimes involves non-trivial (and sometimes even undecidable) calculations, a simpler and more error-prone (but highly successful) solution is used. You may need to add a comment to identify these cases, so check any failing test outputs for instructions on how to resolve incorrectly flagged issues.

Plagiarism

This is an individual assignment. The work must be your own.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as this is considered plagiarism.

If you refer to published work in the discussion of your experiments, be sure to include a citation to the publication or the web link.

“Borrowing” of someone else’s code without acknowledgment is plagiarism. Plagiarism is considered a serious offense at the University of Melbourne. You should read the University code on Academic integrity and details on plagiarism. Make sure you are not plagiarizing, intentionally or unintentionally.

You are also advised that there will be a C programming component (on paper, not on a computer) in the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.

Late Policy

The late penalty is 10% of the available marks for that project for each day (or part thereof) overdue. Requests for extensions on medical grounds will need to be supported by a medical certificate. Any request received less than 48 hours before the assessment date (or after the date!) will generally not be accepted except in the most extreme circumstances. In general, extensions will not be granted if the interruption covers less than 10% of the project duration. Remember that departmental servers are often heavily loaded near project deadlines, and unexpected outages can occur; these will not be considered as grounds for an extension.

Students who experience difficulties due to personal circumstances are encouraged to make use of the appropriate University student support services, and to contact the lecturer, at the earliest opportunity.

Finally, we are here to help! There is information about getting help in this subject on the LMS. Frequently asked questions about the project will be answered on Ed.

Additional Support

Your tutors will be available to help with your assignment during the scheduled workshop times. Questions related to the assignment may be posted on the Ed discussion forum, using the folder tag Assignments for new posts. You should feel free to answer other students' questions if you are confident of your skills.

A tutor will check the discussion forum regularly, and answer some questions, but be aware that for some questions you will just need to use your judgment and document your thinking. For example, a question like, "How much data should I use for the experiments?", will not be answered; you must try out different data and see what makes sense.

If you have questions about your code specifically which you feel would reveal too much of the assignment, feel free to post a private question on the discussion forum.

Have fun!

Code Submission - Deliverable

This code slide does not have a description.

Experimentation

Upload your experimentation file as a PDF.

Submission Certification

In this form you will confirm & certify your submission against the COMP20003 Honor Code and academic integrity of your submission.

COMP20003 Honor Code

We’re committed to the integrity of your work on the COMP20003 course. To do so, every learner-student should:

- Submit their own original work
- Do not share code (solution) with other students

Question 1 *Submitted Oct 20th 2022 at 5:12:39 pm*

Did you enjoy the assignment project?

(This question is for us to understand the usefulness of this project assignment for the future)

- ( ) Yes, a lot!
- (•) Yes
- ( ) More or less
- ( ) Not that much
- ( ) No, I didn't
- ( ) I don't want to participate in this question

Question 2 *Submitted Oct 20th 2022 at 5:12:40 pm*

Do you feel you learn with this assignment project?

(This question is for us to understand the usefulness of this project assignment for the future)

- ( ) Yes, I learnt quite a lot!
- ( ) Yes, I learnt some things
- (•) I learnt some, but not much
- ( ) I learnt almost nothing
- ( ) I don't want to participate in this question

Question 3 *Submitted Oct 20th 2022 at 5:12:46 pm*

Feel free to share any feedback here

We are interested in any constructive negative or positive feedback, both are helpful to reflect what may need to change and what is working. Good humor and politeness are always welcome! ;-) Thanks!

N/A

Question 4 *Submitted Oct 20th 2022 at 5:13:12 pm*

I certify that this was all our original work. If we took any parts from elsewhere, then they were non-essential parts of the project, and they are clearly attributed at the top of the file and in a separate report. I will show I agree to this honor code by typing "Yes":

This declaration is the same as the one in Khan Academy. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us. You are always better off getting a very bad mark (even zero) than risking to go that path, as the consequences are serious for students.

⚠ The project will not be marked unless this question is answered correctly and exactly with "Yes" as required.

Yes