Functions

Procedures

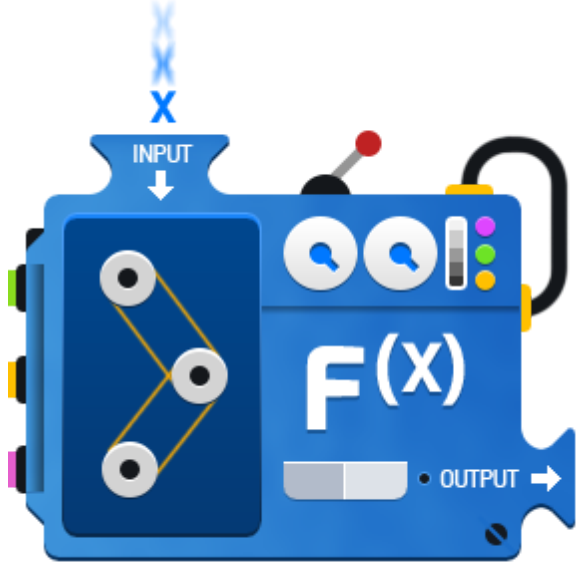
C is a procedural language, i.e. programs are made out of procedures

Traditionally, these procedures are called *functions*

The name isn't really right because procedures can ***do*** things as well as returning a result

Pure functions belong to the Haskell world

What functions do



A function may calculate a result, or manipulate data, or call other functions, or interact with the outside world, e.g. read a file or print out text or control a device

Hello World

It is traditional to start with a Hello World program, though this version is a bit different:

hello.c

```c
/* Example program: say Hi. */
#include <stdio.h>

int main() {
    setbuf(stdout, NULL);
    printf("Hello World!\n");
    return 0;
}
```

Download it, or copy-paste it, or type it, save it as `hello.c`, then compile it, and run it

Compiling and running

To compile and run in a terminal window:

```
$ clang -std=c11 -Wall hello.c -o hello
$ ./hello
Hello World!
$
```

Options `-std=c11` and `-Wall` are essential, and others are desirable, so use `make` and a `Makefile`

See the aside on make to see how the `make` command works

Program files

In the lab, or on your Linux or Mac computer, the compiler produces a new file called `hello`, which is a program ready to run

On Windows, it is called `hello.exe` so you may have to type:

```
$ clang -std=c11 -Wall hello.c -o hello.exe
```

Comments

Let's dissect the Hello World program

The first line is for human readers only, not the computer, to explain what the program is for

```
/* Example program: say Hi. */                    hello.c
```

A program without a comment to explain it is ***rubbish***, and `hello.c` is often copied to start the next program

A `/*...*/` comment can run over many lines

Blank lines are also for human consumption only

Library modules

The next line says "this program needs to use the `stdio` library functions (standard input/output)" because it is going to print something

```
#include <stdio.h>                                    hello.c
```

`stdio.h` is a 'header' file describing the library module, which is included into your program

The angle brackets mean "look in the standard place" (in the labs, that's `/usr/include/stdio.h`)

Functions

The rest of the program is a function

hello.c

```
int main() {
    setbuf(stdout, NULL);
    printf("Hello World!\n");
    return 0;
}
```

Functions provide a way of dividing up a program into pieces (the 'procedural decomposition' design strategy)

Main function

Every C program must have a function called `main`

```
int main() {
    ...
}
```
hello.c

The system calls `main` to start the program running

`main` is a ***rubbish*** name – it ought to be called `run` because you run programs, you don't main them, but it is too late to change the convention

Every useful language has lots of rubbish in it

Defining a function

A function has a return type, a name, arguments, a body

hello.c

```
int main() {
    ...
}
```

The `int` type means "small-ish integers"

The `main` function actually has two arguments:

```
int main(int n, char *args[n]) ...
```

In `main`, you are allowed to ignore them

Buffered output

The name `stdout` refers to the stream of text which the program is going to produce when it runs

```
setbuf(stdout, NULL);                                    hello.c
```

Output is often *buffered* - the text is gathered up until there is a reasonable amount to send efficiently

For `stdout`, that is exceedingly confusing

Many systems detect that `stdout` is going directly to the screen, and switch buffering off, but not all (e.g. MSYS2), so this line explicitly switches off buffering

Why?

The line `setbuf(stdout,NULL);` is not usually needed, and it is not needed in Hello World

But the Hello World program is often used to copy-paste into any program when you start writing it

So it should contain anything you might need

If you are using MSYS2 to write native Windows programs, you need it it at the start of `main`

Otherwise you probably won't need it, so I won't mention it again

Printing

The `printf` function is the commonest one for output

hello.c

```
...
    printf("Hello World!\n");
...
```

It prints a string, but can also print out values

The `\n` at the end is a newline - always include it, otherwise lots of things can go wrong

Add `printf` calls to your program to debug it, if you can't work out what's wrong

Names

`printf` is a rubbish name

`format_and_print`, or `print` (or `write` or `show` to avoid suggesting a printer) would have been better

Many library functions have rubbish names, because:

- nobody had any idea in the early days how important names are - bad names cause bugs
- there was originally a six-letter limit on names

A name should be short, readable and evocative

Returning

A function returns a value at the end (unless its return type is `void`)

hello.c

```
int main() {
    ...
    return 0;
}
```

`main` returns `0` to tell the system it succeeded, or an error value, usually `1`, if it failed

What if you forget to return?

What if you leave out the `return` at the end?

```
int main() {
    setbuf(stdout, NULL);
    printf("Hello World!\n");
}
```

The standard says a function like that returns "undefined" (it returns whatever rubbish value happens to be in the return register)

The `main` function is an exception (for backwards compatibility it returns 0) - so this is legal, but bad

Example: paint

paint.c

```c
/* Find the area of paint I need. */
#include <stdio.h>

// Calculate area of walls and ceiling
int area(int length, int width, int height) {
    int sides = 2 * length * height;
    int ends = 2 * width * height;
    int ceiling = length * width;
    return sides + ends + ceiling;
}

// Find area of paint for my room.
int main() {
    int total = area(5, 3, 2);
    printf("The paint area is %d\n", total);
    return 0;
}
```

Function order

The program has two functions

paint.c

```
...
// Calculate area of walls and ceiling
int area(int length, int width, int height) ...

int main() ...
```

It is important that `area` is defined first, so that the compiler knows about it when `main` calls it

`area` has three arguments

A line starting `//` is a one-line comment

Function calls

You call a function by passing it some values for its arguments, then catching the result that is returned

paint.c

```
int main() {
    ...
    int total = area(5, 3, 2);
    ...
}
```

The arguments must be in the right order, e.g. the height must be last in any call to `area` (think about it!)

Printing again

The `main` function also contains a call to `printf`:

```
printf("The paint area is %d\n", total);  paint.c
```

The `printf` function is the only commonly used function with a variable number of arguments

The `%d` means 'print an integer in decimal format here', and the integer to print is an extra argument to the function

You can use `%i` instead of `%d`, but `%d` is more conventional

Statements

A function contains a sequence of statements, each ending in a semicolon ;

paint.c

```
int area(int length, int width, int height) {
    int sides = 2 * length * height;
    int ends = 2 * width * height;
    int ceiling = length * width;
    return sides + ends + ceiling;
}
```

Say "sides becomes ..." or "set sides equal to ..."

Expressions

Calculations are done using expressions

paint.c

```
... 2 * length * height ...
... 2 * width * height ...
... length * width ...
... sides + ends + ceiling ...
```

It is up to you how you split things up: some people would write:

```
return 2 * length * height + 2 * width * height + length * width;
```

This is less readable: it doesn't explain itself

Integers

So far, for integers, we've been using type `int`, which uses binary with 32 bits, one of which is for the sign

So the range is `-2147483648` to `2147483647`

If you need more, use `long`, with 64 bits, range roughly plus or minus 9 quintillion

Platforms

A platform is a combination of processor, operating system, device drivers, libraries, compiler, run time system, versions and settings of all those, and anything else which affects programs

Usually, we abbreviate by talking about Linux, MacOS, Windows, and the mobile versions Android, iOS, mobile Windows, but these are really extensive families of platforms

Platform differences

Technically `int` is the "best" integer type provided by the processor (was 16 bits, and may become 64 bits)

We are in a happy time where `int` is usually 32 bits and `long` is usually 64

The main exceptions are tiny embedded processors, and native Windows platforms where `long` is 32 bits

Cross-platform programs

In this unit, you ***must*** write cross-platform programs because (a) it is the right thing to do and (b) your submitted programs won't be marked on ***your*** platform

The main techniques for this are sticking rigidly to the language standard, and switching on all compiler error messages, and gaining experience

Minority platforms

It is ***not*** recommended to try to write programs for ***all*** platforms, because the issues on native Windows and tiny processor platforms are too numerous, too restrictive, and too difficult to test

The best approach is (a) write a majority cross-platform program first (b) run through the platform issues one by one (c) use conditional compilation for the fixes

So: let's all assume `int` is 32 bits and `long` is 64

Double

The paint program uses `int`, but we may want non-integer lengths, producing a non-integer area

Just replace `int` by `double` everywhere appropriate

The `double` type is the type of "double precision floating point numbers", and it is the normal type to use for approximate real numbers

In `printf`, use `%f` (floating point) instead of `%d` (decimal)

Example: double paint

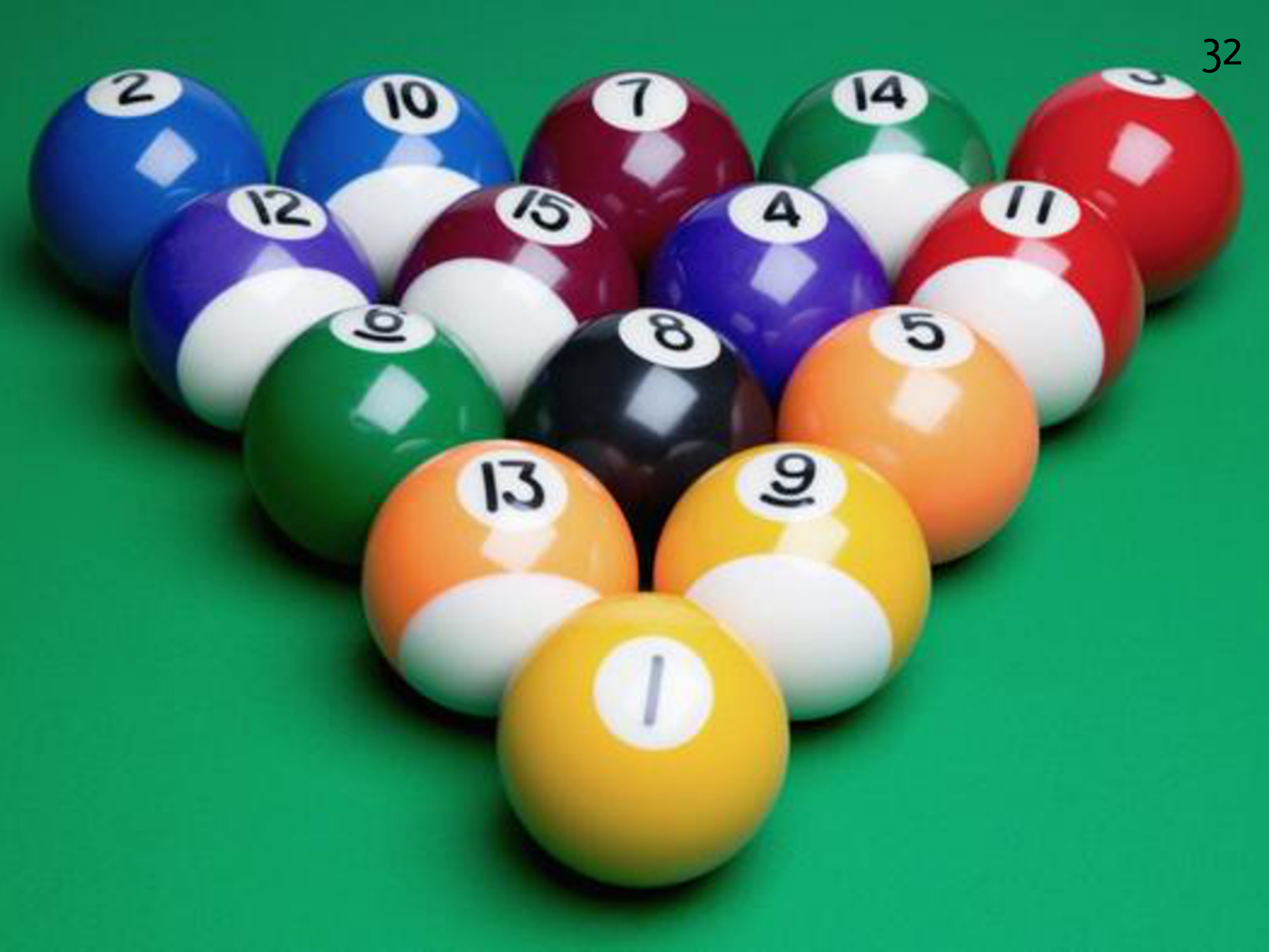
dpaint.c

```c
/* Find the area of paint I need. */
#include <stdio.h>

// Calculate area of walls and ceiling
double area(double length, double width, double height) {
    double sides = 2 * length * height;
    double ends = 2 * width * height;
    double ceiling = length * width;
    return sides + ends + ceiling;
}

// Find area of paint for my room.
int main() {
    double total = area(5, 3, 2);
    printf("The paint area is %f\n", total);
    return 0;
}
```

Integer constants 2, 5,... get converted to `double`


2
10
7
14
12
15
4
11
8
5
13
9
1

Example: triangle numbers

The n^{th} triangle number is the sum of the numbers from *1* to *n*

sum.c

```c
/* Find the n'th triangle number. */
#include <stdio.h>

// Find the sum of the numbers from 1 to n.
int sum(int n) {
    if (n == 1) return 1;
    else return n + sum(n-1);
}

int main() {
    int t10 = sum(10);
    printf("The 10th triangle number is %d\n", t10);
    return 0;
}
```

The sum function

The important part of the program is the sum function:

```
// Find the sum of the numbers from 1 to n. sum.c
int sum(int n) {
    if (n == 1) return 1;
    else return n + sum(n-1);
}
```

It has an *argument* variable n

The argument n is *local*, it is created at the start of a call, and destroyed when the function returns

The function is *recursive*, i.e. it calls itself

A row of friends

To get the hang of recursion, imagine a row of 10 friends who cooperate in solving the problem

Each friend has a copy of the instructions:

sum.c

```
int sum(int n) {
    if (n == 1) return 1;
    else return n + sum(n-1);
}
```

The `main` function calls `sum(10)`, which is like handing the `sum(10)` problem to one friend, let's say Alice who writes `n = 10` on her piece of paper

The next friend

Alice obeys this instruction:

```
return n + sum(n-1);                                    sum.c
```

This involves a function call `sum(9)`, which is like handing the `sum(9)` problem to the next friend, let's say Bob, who writes `n = 9` on his piece of paper

The last friend

The requests go down the line until the problem `sum(1)` is handed to Joe who obeys this instruction:

```
return 1;                                                    sum.c
```

Irene receives `1` and adds it to the number on the paper, `2`, and returns `3`

Back up the line

The answers go back up the line

Friend Henry receives `3` and returns `6`

Friend Grace receives `6` and returns `10`

...

Friend Alice receives `45` and returns `55`

Local variables

The argument variable n is local

It is created when the call is made, set to the number passed in the call, and it lasts until the call returns

It is like a friend's piece of paper

It can't be accessed from outside the function – it belongs to the function – you can think of it as 'trapped' by the curly brackets

Call stack

A processor has a *call stack*, containing *stack frames*, like a pile of pieces of paper with local variables written on, one for each function call which is in progress

Later, when we get to pointers, we will have a look at its layout

It is very efficient, especially since call and return instructions are built into the processor

Termination

It is a good thing the `sum` function doesn't ***always*** call itself

Otherwise, there would be an unlimited chain of calls (often called an 'infinite loop') and the program would keep going until it ran out of memory

Recursion always needs a termination condition ('get out clause')

Is recursion important?

Recursion is much rarer in C than in Haskell, because loops are often used instead:

```
int sum(int n) {
    int result = 0;
    for (int i=1; i<=n; i++) result = result + i;
    return result;
}
```

But occasionally, recursion is essential or natural, as the clearest solution to a problem

Design

The C language hasn't changed all that much over time

The way C programmers design programs has changed

To illustrate, on the next couple of slides, there is a before and after example of a prime number program in the old style and in the new style

You don't need to understand it all

The difference doesn't matter much for small programs, but becomes crucial for bigger ones

Old primes

oldprimes.c

```
/*
 * This program generates prime numbers up to a user specified
 * maximum. The algorithm used is the Sieve of Eratosthenes.
 *
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
 * d. c. 194, Alexandria. The first man to calculate the
 * circumference of the Earth. Also known for working on
 * calendars with leap years and ran the library at Alexandria.
 *
 * The algorithm is quite simple. Given an array of integers
 * starting at 2. Cross out all multiples of 2. Find the next
 * uncrossed integer, and cross out all of its multiples.
 * Repeat untilyou have passed the square root of the maximum
 * value.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
 * From book "Clean Code", adapted from Java by Ian Holyer
 * Compile with: clang -o oldprimes oldprimes.c -lm
 * Run with: ./oldprimes 10
 */
```

Old primes 2

oldprimes.c

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

int main(int argc, char **argv) {
  if (argc < 2) { printf("Please give a maximum number\n"); exit(1); }
  // maxValue is the generation limit.
  int maxValue = atoi(argv[1]);
  if (maxValue >= 2) { // the only valid case
    // declarations
    int s = maxValue + 1; // size of array
    int f[s];
    int i;
    // initialize array to true.
    int false = 0, true = 1;
    for (i = 0; i < s; i++)
      f[i] = true;
    // get rid of known non-primes
    f[0] = f[1] = false;
```

Old primes 3

oldprimes.c

```
    // sieve
    int j;
    int root = (int)sqrt((double)s);
    for (i = 2; i < root + 1; i++) {
      if (f[i]) { // if i is uncrossed, cross its multiples.
        for (j = 2 * i; j < s; j += i)
        f[j] = false; // multiple is not prime
      }
    }
    for (i = 0; i < s; i++) {
      if (f[i]) printf("%d\n", i);
    }
    return 0;
  }
  else // maxValue < 2
    return 1; // return program failure if bad input.
}
```

New primes

primes.c

```c
/* Generate primes up to a maximum using
http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.
Compile with: clang -std=c11 -Wall primes.c -lm -o primes
Run with: ./primes 10
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <stdbool.h>

// Extract 'max' from the command line, add one to make array size.
int findSize(int n, char *args[n]) {
  if (n != 2) {
    fprintf(stderr, "Use: ./primes max\n");
    exit(1);
  }
  return atoi(args[1]) + 1;
}

// Clear the array of booleans, so only 0 and 1 are crossed out.
void uncrossAll(int size, bool crossedOut[size]) {
  for (int i=0; i<size; i++) crossedOut[i] = false;
  crossedOut[0] = crossedOut[1] = true;
}
```

New primes 2

primes.c

```
// Cross out multiples of a number n
void crossOutMultiples(int size, bool crossedOut[size], int n) {
  for (int m = 2*n; m < size; m = m + n) crossedOut[m] = true;
}

// See wikipedia: every composite has a prime factor <= its square root
// so we only need to cross out multiples of numbers up to sqrt(size)
int findIterationLimit(int size) {
  double root = sqrt((double)size);
  return (int) root;
}

// Cross out all composite numbers
void crossOutComposites(int size, bool crossedOut[size]) {
  int limit = findIterationLimit(size);
  for (int i = 2; i <= limit; i++) {
    if (! crossedOut[i]) crossOutMultiples(size, crossedOut, i);
  }
}

// Follow the algorithm
void generatePrimes(int size, bool crossedOut[size]) {
  uncrossAll(size, crossedOut);
  crossOutComposites(size, crossedOut);
}
```

New primes 3

primes.c

```
// Print the un-crossed-out numbers
void printPrimes(int size, bool crossedOut[size]) {
  for (int i = 2; i < size; i++) {
    if (! crossedOut[i]) printf("%d\n", i);
  }
}

void test() {
    bool expected[12] = {1,1,0,0,1,0,1,0,1,1,1,0};
    bool crossedOut[12];
    generatePrimes(12, crossedOut);
    for (int i = 0; i < 12; i++) if (crossedOut[i] != expected[i]) {
        fprintf(stderr, "Wrong result for %d\n", i);
        exit(1);
    }
    printf("All tests pass.");
}

// Run
int main(int n, char *args[n]) {
  if (n == 1) test();
  int size = findSize(n, args);
  bool crossedOut[size];
  generatePrimes(size, crossedOut);
  printPrimes(size, crossedOut);
}
```

Modern design

The modern design style is:

- tiny functions, visible at a glance
- each has one clear responsibility
- each is quite readable
- calculation is separated from input/output
- calculation functions are autotested

Prime improvements

The program as a whole is more readable

The functions are self-documenting at the 'how' level

The comments are brief, external, adding the 'why'

Commenting-out can be used during development

Each function is short enough to ***see*** it is correct

The functions can be developed one by one

Automatic testing adds confidence

Commenting out

Suppose you change something radical, so all the functions in the program need to be changed

You can surround them all with a /*...*/ comment, then move them out of the comment one by one

But /*...*/ comments don't nest, so this doesn't work if you use /*...*/ comments before function definitions

So using one-line // comments before functions makes commenting-out easy

Side Effects

A function can do these things

1. return a result
2. change data passed as arguments
3. input or output
4. change global variables

A function that only does 1 is ***pure***, see Haskell

Functions that do 2, 3 or 4 are said to have side effects

In C, 2 is normal, but it is best to separate out 3 and avoid 4, because 1 and 2 can be auto-tested