Introduction to the C programming language

Wes Armour

Oxford e-Research Centre,
Department of Engineering Science

Learning outcomes

In this lecture you will learn about:

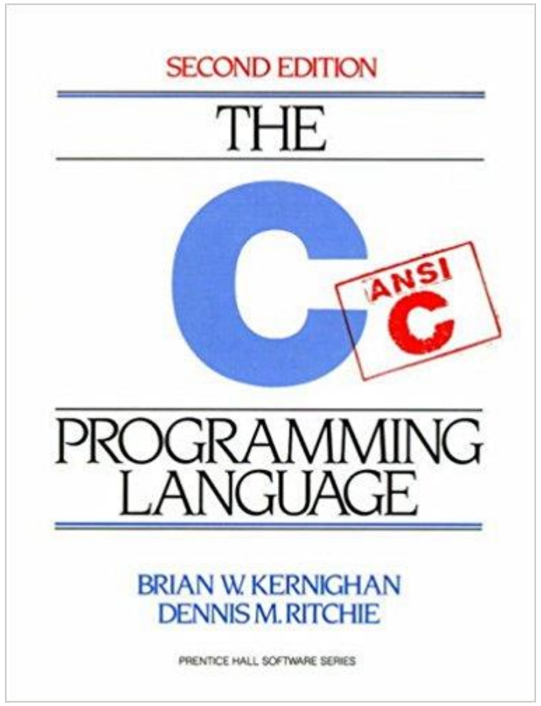
- High level computer languages.
- The basic components of a C computer program.
- How data is stored on a computer.
- The difference between statements and expressions.
- What operators and functions are.
- How to control basic input and output.
- Finally how to write a basic C program.

A brief introduction

The C programming language was devised by Dennis Ritchie at Bell labs in 1972 (yes, it’s predecessor was B!).

C is a high-level programming language, meaning that it is possible to express several pages of machine code in just a few lines of C code.

Other examples of high-level languages are BASIC, C++, Fortran and Pascal. They are so called because they are closer to human language than machine languages.



A brief introduction

Such high-level languages allow a programmer to write programs that are independent of particular types of computer. This is called portability.

Portability can be aided by using an agreed standard when writing your program (such as C99).

A compiler (such as gcc or icc) is used to convert your high-level language program into machine code that can be executed on a computer.

HLL

```
int square(int num) {

    return num * num;

}
```

Assembly Code

```
push rbp #2.21
mov rbp, rsp #2.21
sub rsp, 16 #2.21
mov DWORD PTR [-16+rbp], edi #2.21
mov eax, DWORD PTR [-16+rbp] #3.18
imul eax, DWORD PTR [-16+rbp] #3.18
leave #3.18
ret #3.18
```

Machine Code

```
01110000011010001010100100100101001
00011111111111101010111100111001010
10101010011111111111101010100101010
10101010101010101001010101010100101
```

https://godbolt.org/

The components of a C program

All C programs have some common elements.

- The `#include` directive tells the compiler to include other files stored on your HDD into your C program. These files will include information that does not change between programs that your program can use. On the right we include the standard input and output library.

- The `main()` function is the only component that has to be included in every C program. It is followed by a pair of braces: `{ }`

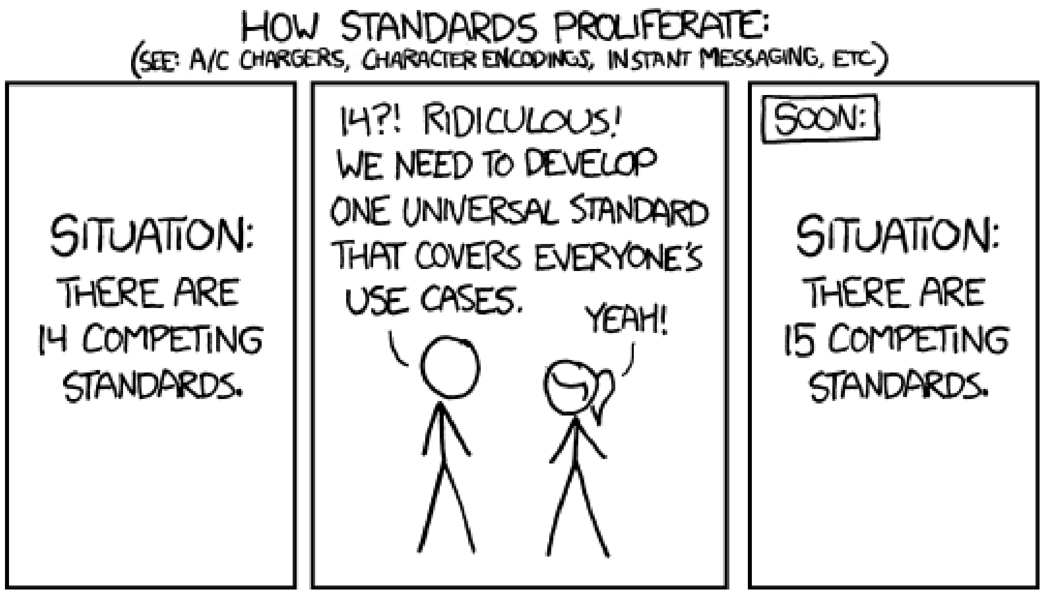
- The `return()` statement returns values from a function. Within the `main()` function `return()` can be used to tell the operating system (in our case Linux) whether our code completed successfully.

```
#include <stdio.h>

int main() {

    return(0);

}
```

The C standard library

The C programing language has a rich set of tools that you can use in your code to make it portable and easer to write.

Using these tools (contained in the C standard library) will save you time, effort and make your code more understandable to others.


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

https://xkcd.com/927/

Standard header files

The list on the right describes some of the more common *header files*.

These are included into your code in the same way as we included the standard input/output header, using the `#include` directive.

These tools can save you lots of time, for example you could use the random number generator included in `<stdlib.h>` rather than writing your own.

Name	*Description*
<assert.h>	Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.
<complex.h>	A set of functions for manipulating complex numbers.
<errno.h>	For testing error codes reported by library functions.
<math.h>	Defines common mathematical functions.
<stdbool.h>	Defines a boolean data type.
<stdio.h>	Defines core input and output functions
<stdlib.h>	Defines numeric conversion functions, pseudo-random numbers generation functions, memory allocation, process control functions
<string.h>	Defines string handling functions.
<time.h>	Defines date and time handling functions

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Storing data

Our previous example program didn't do anything though. To make this more useful we need to store and manipulate data.

A computer stores data as strings of zeros and ones: 100101111001…

The smallest element of data storage on a computer is a bit, it can be two things, a zero or a one.

Eight bits combine to produce a byte. If each bit in a byte can be either a zero or a one then a byte can represent:

$$2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^8 = 256$$

unique "things". For example these "things" could be the numbers from -128 to 127.

In the C language this would be called a char variable.

```
#include <stdio.h>

int main() {

    char x;

    return(0);

}
```

Variables and Constants

The C language has different *numeric data types* that you can use in your program depending on what type of number you need to store or manipulate.

This is useful for many reasons. For example if our program only operates on small numbers (-128 to 127) then we can use a char which takes up a single byte of memory rather than using a long integer that might take 4 bytes or 8 bytes of memory space.

Along with this we might want to work on decimal numbers. In C, real numbers are stored as floating point variables, these have a fractional part. Integers are stored as integer numbers and have no fractional part.

Variables with more bytes require more computational operations to manipulate them, so this will slow down program execution.

Data Type	***Keyword***	***Minimum Bytes Required***	***Minimum Range***
Character	char	1	–128 to 127
Short integer	short	2	–32767 to 32767
Integer	int	4	–2,147,483,647 to 2,147,438,647
Long integer	long	4	–2,147,483,647 to 2,147,438,647
Unsigned character	unsigned char	1	0 to 255
Unsigned short integer	unsigned short	2	0 to 65535
Unsigned integer	unsigned int	4	0 to 4,294,967,295
Unsigned long integer	unsigned long	4	0 to 4,294,967,295
Single-precision floating-point	float	4	1.2E–38 to 3.4E38
Double-precision floating-point	double	8	2.2E–308 to 1.8E308

Variables and Constants

The amount of bytes needed to store any particular numeric data type in memory can vary between computer architectures.
C provides a useful function `sizeof()` to determine this.

Numeric data types can either be a *variable* or a *constant* in your C program.

Variables and *literal constants* can change during your program execution, *symbolic constants* cannot.

Literal constants are defined by assigning a number to the constant:

```
float radius = 10;
```

A symbolic constant can be defined using two methods. The first is by using `#define`, the other by using the `const` keyword.

```
const float radius = 10;
```

```
#include <stdio.h>

#define PI 3.14

int main() {

    float radius = 10.0;
    float area;

    area = PI * radius * radius;

    return(0);

}
```

Statements

A *statement* in C is a command that instructs the computer to do something.

An example of a statement is:

```
 area = PI * radius * radius;
```

This tells the computer to multiply PI by radius and the multiply the results of this by radius again. The result of the multiplications is then assigned to the variable `area`

All statements in C are terminated with a semi-colon and all white space (tabs, spaces and blank lines) are ignored by the compiler (apart from within a *string* – more later).

This allows for lots of freedom in formatting your C program, however to ensure that your code is easy to work with and reusable by others it is important to ensure that it is easy to read and understand.

```
area = PI * radius * radius;
```

Is equivalent to

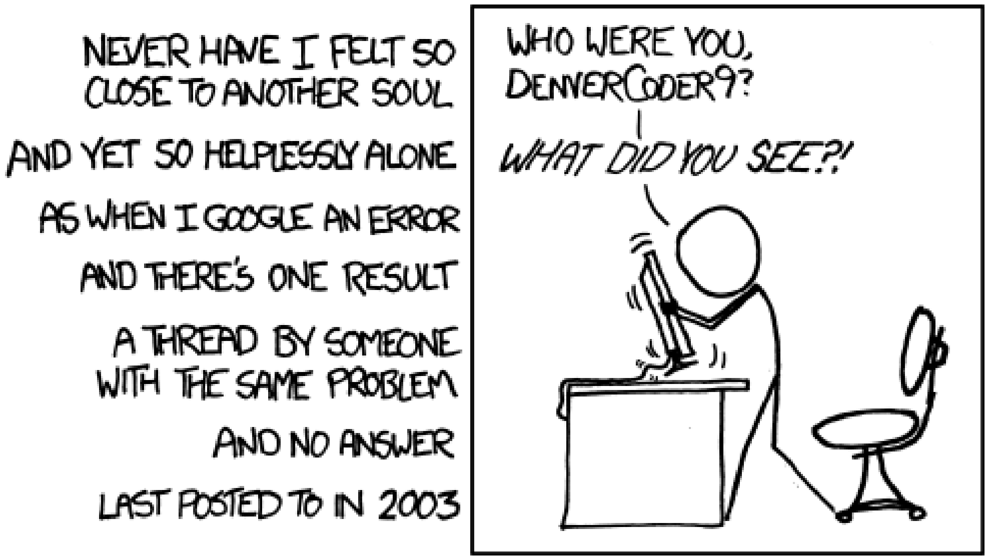
```
area =
       PI *
          radius
                     * radius
             ;
```

Comments

To ensure that your code is easy to read and understand it's important to write your code in a readable and maintainable way.

Adding comments to your code will help others understand your code when they come to work on it. A comment is text that you or other programmers can read but is ignored by the compiler. A good code has more comments than source code.

// starts a single line comment
/* Starts a multiline comments and is ended by */


NEVER HAVE I FELT SO CLOSE TO ANOTHER SOUL
AND YET SO HELPLESSLY ALONE
AS WHEN I GOOGLE AN ERROR
AND THERE'S ONE RESULT
A THREAD BY SOMEONE WITH THE SAME PROBLEM
AND NO ANSWER
LAST POSTED TO IN 2003
WHO WERE YOU, DENVERCODER9?
WHAT DID YOU SEE?!

https://xkcd.com/979/

Naming

Using variable names that are meaningful is a useful thing to do, for example:

```
area = PI * radius * radius;
```

is easy to understand, whereas

```
one = c * two * two;
```

tells us less about what you want your code to achieve. Take time to name variables in a meaningful and intuitive way.

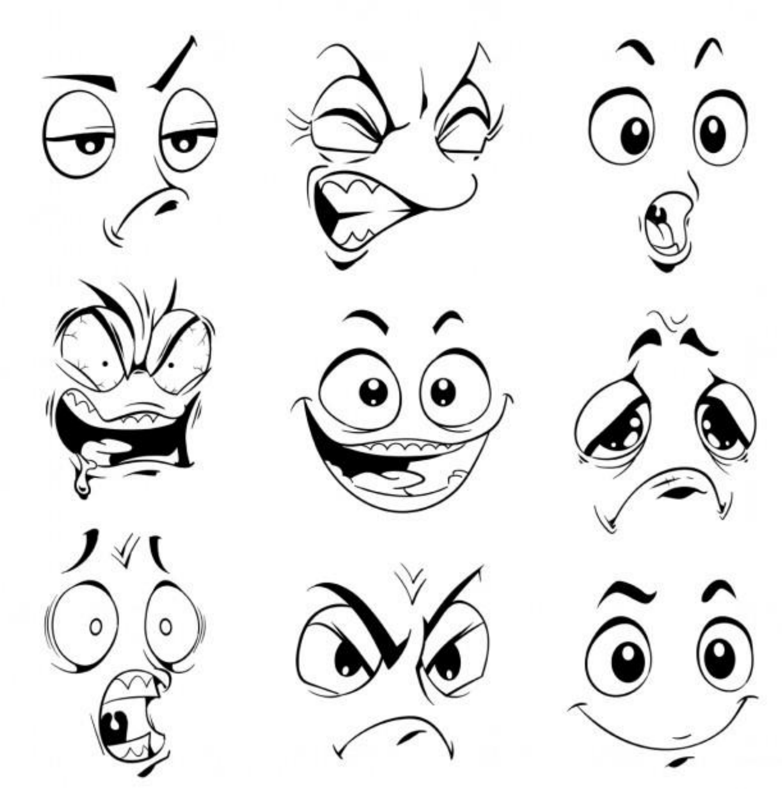
```
/* This is a C program that
   calculates the area of a circle.
   Written by Wes
   wes.armour@eng.ox.ac.uk
   06/05/18
*/

//Include standard IO library
#include <stdio.h>

// Define a symbolic constant, pi.
#define PI 3.14

// The main body of the program
int main() {

    // Define variables
    float radius = 10.0;
    float area;

    // Calculate the area
    area = PI * radius * radius;

    return(0);
}
```

Expressions

An *expression* in C is any statement that evaluates to a numeric value. In the most basic form this could just be our previous example of using a symbolic constant, `PI`

We use the *assignment operator* `(=)` to assign the result of our expression to a variable:

```
 variable = expression;
```



https://www.freepik.com/free-photos-vectors/people   Created by Ddraw - Freepik.com

Expressions

However statements can become complicated very quickly. Consider summing the area of three circles:

```
a = p*a*a+p*b*b+p*c*c;
```

When trying to evaluate the above where does the computer start with such an expression? What stops the computer evaluating `a+p` first?

Several things can help us ensure that we get the computer to do the right thing, separating our expression, using *brackets* and *operator precedence*.

```
#include <stdio.h>

#define PI 3.14

int main() {

    float radius_one   = 5.0;
    float radius_two   = 10.0;
    float radius_three = 15.0;
    float area_one, area_two, area_three;
    float total_area;

    area_one = PI * radius_one * radius_one;
    area_two = PI * radius_two * radius_two;
    area_three = PI * radius_three * radius_three;

    total_area = area_one + area_two + area_three;

    return(0);

}
```

Operators

An *operator* in C is a symbol that instructs the computer to perform an operation on an *operand.*

Precedence tells the computer which operation should be performed first in an expression. Returning to our example of calculating the area of a circle:

```
area = PI * radius * radius;
```

We see that multiply (*) has a precedence of 3, whereas assignment (=) has a lower precedence of 7, meaning multiplication is carried out then the assignment.

The order in which the multiplication is carried out is determined by the *Associativity* of multiplication. We see that this is Left-to-right. So the order would be:

```
(PI * radius) * radius
```

Precedence	*Operator*	*Description*	*Associativity*
1	++	Postfix increment	Left-to-right
	--	Postfix decrement	
	()	Function call	
2	++	Prefix increment	Right-to-left
	--	Prefix decrement	
	sizeof	Size-of	
3	*	Multiplication	Left-to-right
	/	Division	
	%	Modulo (remainder)	
4	+	Addition	Left-to-right
	-	Subtraction	
5	<	Less than	Left-to-right
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
6	==	Equal to	Left-to-right
	!=	Not equal to	
7	=	Direct assignment	Right-to-left
	+=	Assignment by sum	
	-=	Assignment by difference	
	*=	Assignment by product	
	/=	Assignment by quotient	

Functions

A *function* is an independent piece of C code that performs a specific task. The function may or may not return a value to the calling code. For example it might calculate the area of a circle, or it might print a message to the screen.

- A function has a unique name
- A function is independent of other parts of your code, so it is self contained.
- A function performs a specific task in your code.
- A function may or may not have a return value.

On the right we see how we can use a function in our example code that calculates and sums the area of three circles.

```
#include <stdio.h>

#define PI 3.14

float area_of_circle(float radius);

int main() {

    float radius_one   = 5.0;
    float radius_two   = 10.0;
    float radius_three = 15.0;
    float area_one, area_two, area_three;
    float total_area;

    area_one = area_of_circle(radius_one);
    area_two = area_of_circle(radius_two);
    area_three = area_of_circle(radius_three);

    total_area = area_one + area_two + area_three;

    return(0);

}

float area_of_circle(float radius) {

    float area = PI * radius * radius;

    return area;
}
```

Functions

Looking at our example in a little more detail.

`float area_of_circle(float radius);`

Is the *function prototype* this tells the compiler that a function called `area_of_circle` will be used later in the code and it will take a float as an argument and return a float.

We then see the function being called to calculate `area_one`, `area_two` and `area_three`.

Finally outside of the `main() { }` function we see the *function definition.* The first line is the *function header* and this is exactly the same as the *function prototype* (without the semicolon). It defines the functions name, the *parameter list* (variables and their types that are passed to the function by the calling code), whether it returns a value (in this case the *return type* is a float) and then what it does.

```
#include <stdio.h>

#define PI 3.14

float area_of_circle(float radius);

int main() {

    float radius_one   = 5.0;
    float radius_two   = 10.0;
    float radius_three = 15.0;
    float area_one, area_two, area_three;
    float total_area;

    area_one = area_of_circle(radius_one);
    area_two = area_of_circle(radius_two);
    area_three = area_of_circle(radius_three);

    total_area = area_one + area_two + area_three;

    return(0);

}

float area_of_circle(float radius) {

    float area = PI * radius * radius;

    return area;
}
```

Arrays

Our previous example code could be come very cumbersome and difficult to maintain if we wanted to calculate the area of thousands of circles.

C has *arrays* to help with this. An array is a indexed group of data storage, all of the same type.

C arrays are indexed from 0 to n-1, where n is the number of elements in the array.

An array is defined in the following way:

```
float radius[3];
```

Here we define an array called radius that has 3 elements. It can be initialised using braces.

```
float radius[3] = {5, 10, 15};
```

```
#include <stdio.h>

#define PI 3.14

float area_of_circle(float radius);

int main() {

    float radius[3] = {5.0, 10.0, 15.0};
    float area[3];
    float total_area;

    area[0] = area_of_circle(radius[0]);
    area[1] = area_of_circle(radius[1]);
    area[2] = area_of_circle(radius[2]);

    total_area = area[0] + area[1] + area[2];

    return(0);

}

float area_of_circle(float radius) {

    float area = PI * radius * radius;

    return area;
}
```

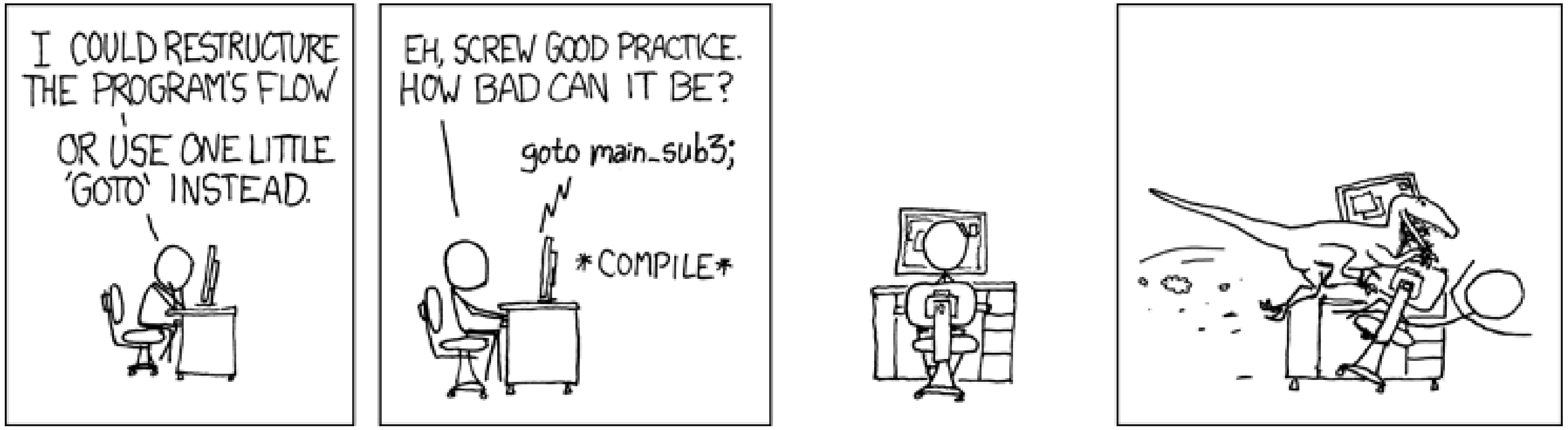
Break

Program control

To solve a particular problem your C program might need to take different execution paths. These might depend on different inputs.

For example we could construct a program that calculates the area or circumference of a circle depending on what the user requests.

C has various statements that give the programmer control over the flow of execution in their program. However it's important to use these sensibly and not create unmaintainable "spaghetti" code.


I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE

https://xkcd.com/292/

Relational operators

The C language has a set of *relational operators* that are used to compare expressions.

For example we could check to see if our radius is less than or equal to 10:

```
radius <= 10.0
```

If our radius is less than or equal to 10 the above expression evaluates to true (represented by 1).
If our radius is greater than 10 then the above expression evaluates to false (represented by 0).

Precedence	*Operator*	*Description*	*Associativity*
5	<	Less than	Left-to-right
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
6	==	Equal to	Left-to-right
	!=	Not equal to	

Logical operators

What if we need to compare more than one expression at the same time?
C has logical operators to help with this.

Logical operators allow us to combine two or more relational expressions into one single expression.

For example we could check to see if our radius is greater than 5 and less than or equal to 10:

```
radius > 5.0 && radius <= 10.0
```

Precedence	*Operator*	*Description*	*Associativity*
2	!	Logical NOT	Right-to-left
11	&&	Logical AND	Left-to-right
12	||	Logical OR	Left-to-right

Program control – if statement

The `if` statement evaluates an expression, if the expression evaluates to true (1) then the code following the `if` statement executes.

The example on the left checks to see if we have a radius less than or equal to 10, if we do then it calculates the area of a circle.

```
#include <stdio.h>

#define PI 3.14

int main() {

    float radius = 10.0;
    float area;

    if(radius <= 10.0) {
        area = PI * radius * radius;
    }

    return(0);

}
```

Program control – if statement

The `if` statement also has an `else` clause. The else clause executes when the expression evaluated by the if statement evaluates to false.

An example of this is given on the right. Here we see that when we have a radius less than or equal to 10.0 our code calculates the area of a circle. For a radius greater than 10 the code calculates the circumference. In this example `radius = 11.0`, so the code would calculate the circumference.

The example on the right is binary in the sense that the result gives two different outcomes dependent on whether our expression is true or false. This can be expanded to give many different outcomes by using `else if` (see practical 1).

```
#include <stdio.h>

#define PI 3.14

int main() {

    float radius = 11.0;
    float area;
    float circumference;

    if(radius <= 10.0) {
        area = PI * radius * radius;
    } else {
        circumference = 2.0 * PI * radius;
    }

    return(0);
}
```

Program control – for statement

The `for` statement (*often called the for loop*) executes a block of statements a certain number of defined times. It has the following structure:

for(start point; relational expression; increment)
    statement;

Our example on the right executes as follows:

1. start point is an integer index that is set to zero (`i = 0`).
2. The relational expression is evaluated, our index `i` is less than 3 (`i<3`) so the condition is true (if this evaluates to false the for loop terminates).
3. Statement then executes (e.g. the area of a circle is calculated).
4. Next the index `i` is incremented by one (`i++`) and execution returns to step 2.

```
#include <stdio.h>

#define PI 3.14

float area_of_circle(float radius);

int main() {

    int i;
    float radius[3] = {5.0, 10.0, 15.0};
    float area[3];
    float total_area;

    for(i=0; i<3; i++) {
        area[i] = area_of_circle(radius[i]);
    }

    total_area = 0.0;
    for(i=0; i<3; i++) {
        total_area += area[i];
    }

    return(0);
}

float area_of_circle(float radius) {

    float area = PI * radius * radius;

    return area;
}
```

Program control – for statement

The diagram on the right represents the `for` loop as a flow diagram.

Consider the following for loop:

```
    for(A; B; C)
        D;
```

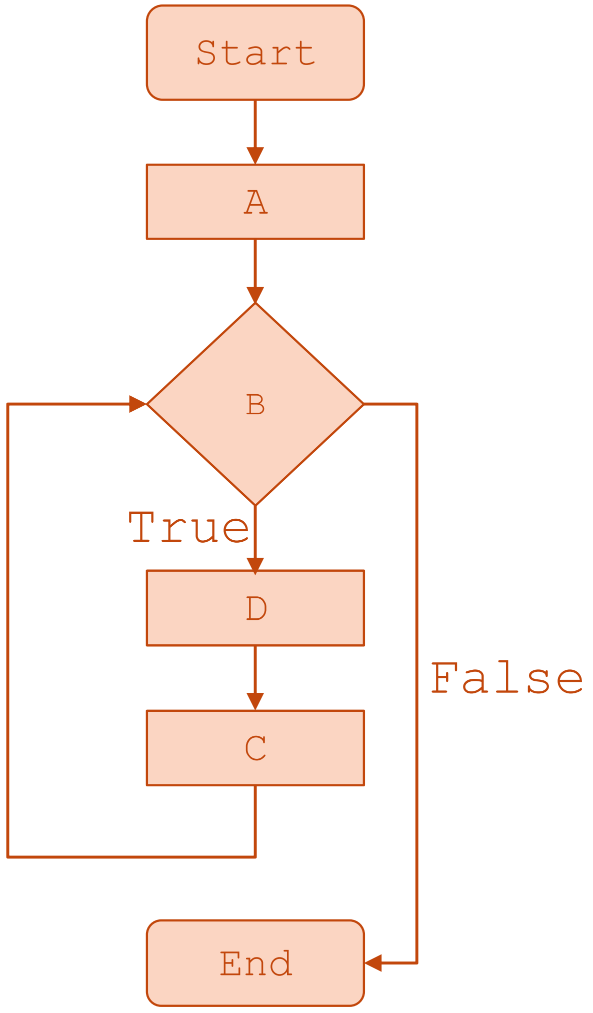
The initial starting point A is evaluated.

Then the relational condition B is evaluated.

If B evaluates to true then statement D is executed. If B evaluates to false then the for loop ends.

Once statement D has executed, the increment C is evaluated.

The loop then returns to evaluating the relational condition B.



Program control – while statement

The `while` statement (*often called the while loop*) executes a block of statements while a certain condition is true. It has the following structure:

while(relational expression)
            statement;

Our example on the right executes as follows:

1. We fist set our integer index i to zero (`i = 0`).
2. The relational expression is evaluated, our index `i` is less than 3 (`i<3`) so the condition is true (if this evaluates to false the while loop terminates).
3. Statement then executes (e.g. the area of a circle is calculate).
4. Next the index `i` is incremented by one (`i++`) and execution returns to step 2.

```
#include <stdio.h>

#define PI 3.14

float area_of_circle(float radius);

int main() {

    int i;
    float radius[3] = {5.0, 10.0, 15.0};
    float area[3];
    float total_area;

    i=0;
    while(i<3) {
        area[i] = area_of_circle(radius[i]);
        i++;
    }

    total_area = 0.0;
    for(i=0; i<3; i++) {
        total_area += area[i];
    }

    return(0);
}

float area_of_circle(float radius) {
    float area = PI * radius * radius;
    return area;
}
```

Program control – while statement

The diagram on the right represents the `while` loop as a flow diagram.

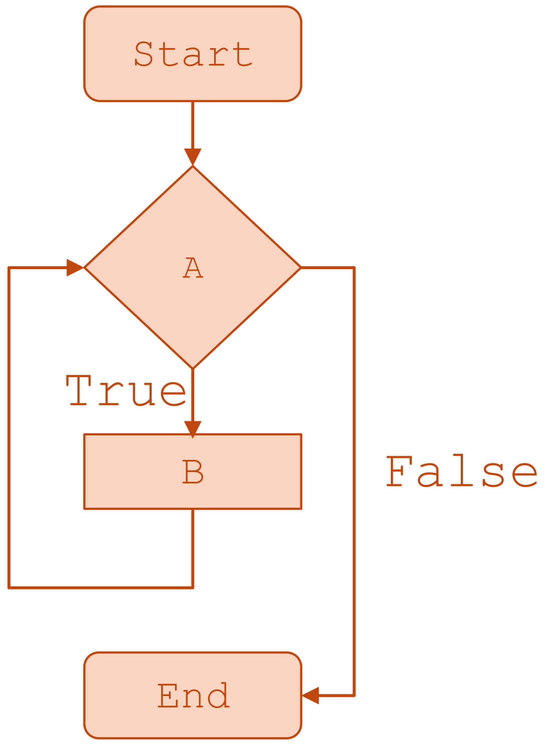
Consider the following while loop:

```
while(A)
    B;
```

The relational condition A is evaluated.

If A evaluates to true then statement B is executed. If A evaluates to false then the while loop ends.

Once statement B has executed, the loop evaluates relational condition A again.



Inputs and Outputs

Many years ago computers were programmed using a series of punch cards (right)!

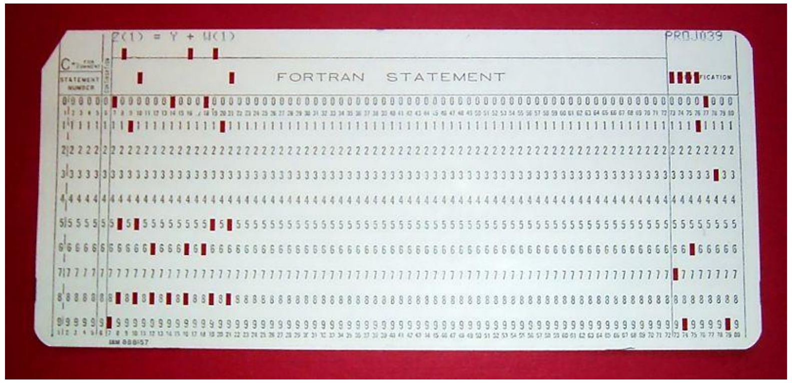
Fortunately the C language has many functions to help with input and output.

In this lecture we will look at two commonly used functions.

To print information out to the screen we will use `printf()`

To read information from the keyboard we will use `scanf()`

These are contained in the stdio.h library.



Arnold Reinhold https://commons.wikimedia.org/wiki/File:FortranCardPROJ039.agr.jpg

Inputs and Outputs

To format input and output C uses conversion specifiers and escape sequences.
The use of these will become clear in the following slides.

Escape sequence	Description
\b	Backspace
\n	Newline
\t	Horizontal Tab
\v	Vertical Tab
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark

Conversion specifier	Type converted	Description
%c	char	char single character
%d	int	signed integer
%ld	long int	long signed integer
%f	float, double	float or double signed decimal
%s	char[]	sequence of characters
%u	int	unsigned integer
%lu	long int	long unsigned integer

Inputs and Outputs - printf

The keen eyed amongst you will have noticed an issue with our example program. Although we calculate a sum of areas of circles, we never actually get the result out of our program.

We can do this using the `printf()` statement. In our example program on the right you can see that we've added a printf() statement.

It works as follows…

```
#include <stdio.h>

#define PI 3.14

float area_of_circle(float radius);

int main() {

    int i;
    float radius[3] = {5.0, 10.0, 15.0};
    float area[3];
    float total_area;

    i=0;
    while(i<3) {
        area[i] = area_of_circle(radius[i]);
        i++;
    }

    total_area = 0.0;
    for(i=0; i<3; i++) {
        total_area += area[i];
    }
    printf("\nTotal area is:\t%f\n", total_area);
    return(0);
}

float area_of_circle(float radius) {
    float area = PI * radius * radius;
    return area;
}
```

Inputs and Outputs - printf

We tell printf to print a string to the monitor:
`“\nTotal area is:\t%f\n”`

We start a new line using the “\n” escape sequence.

We the print the characters *“Total area is:”*

We use a tab to neatly separate our words from our output number using the “\t” escape sequence.

We use the conversation specifier for a float “%f” to output a float.

We tell printf that the float to output is `total_area`

```
#include <stdio.h>

#define PI 3.14

float area_of_circle(float radius);

int main() {

    int i;
    float radius[3] = {5.0, 10.0, 15.0};
    float area[3];
    float total_area;

    i=0;
    while(i<3) {
        area[i] = area_of_circle(radius[i]);
        i++;
    }

    total_area = 0.0;
    for(i=0; i<3; i++) {
        total_area += area[i];
    }
    printf(“\nTotal area is:\t%f\n”, total_area);
    return(0);
}

float area_of_circle(float radius) {
    float area = PI * radius * radius;
    return area;
}
```

Inputs and Outputs - scanf

You will have also noticed that we *hardcode* the radii into our code {5.0,10.0,15.0}

So if we wanted to use different radii we would need to change these values in our source code and then recompile. That’s not very efficient or portable.

We can use the `scanf()` statement to read three different values, meaning our code will work for any combination of radii.

```
#include <stdio.h>
#define PI 3.14

float area_of_circle(float radius);

int main() {
    int i=0;
    float radius[3];
    float area[3];
    float total_area;

    printf(“\nEnter three radii:\t”);
    scanf(“%f %f %f”, &radius[0], &radius[1], &radius[2]);

    while(i<3) {
        area[i] = area_of_circle(radius[i]);
        i++;
    }

    total_area = 0.0;
    for(i=0; i<3; i++) {
        total_area += area[i];
    }
    printf(“\nTotal area is:\t%f\n”, total_area);
    return(0);
}

float area_of_circle(float radius) {
    float area = PI * radius * radius;
    return area;
}
```

Inputs and Outputs - scanf

The example code on the right uses the `scanf()` statement to read three floats into our code and then uses these values to calculate our total area as before.

Once we execute our code it will print the message “Enter three radii:” to the monitor and then wait at the scanf() statement for the user to enter three values and press return.

scanf() reads the three values entered by the user and stores them in radius[0], radius[1] and radius[2] respectively.

```
#include <stdio.h>
#define PI 3.14

float area_of_circle(float radius);

int main() {
    int i=0;
    float radius[3];
    float area[3];
    float total_area;

    printf(“\nEnter three radii:\t”);
    scanf(“%f %f %f”, &radius[0], &radius[1], &radius[2])

    while(i<3) {
        area[i] = area_of_circle(radius[i]);
        i++;
    }

    total_area = 0.0;
    for(i=0; i<3; i++) {
        total_area += area[i];
    }
    printf(“\nTotal area is:\t%f\n”, total_area);
    return(0);
}

float area_of_circle(float radius) {
    float area = PI * radius * radius;
    return area;
}
```

What have we learnt?

In this lecture you have learnt about the basic building blocks of a C program.

You have learnt about standard libraries, expressions and statements.

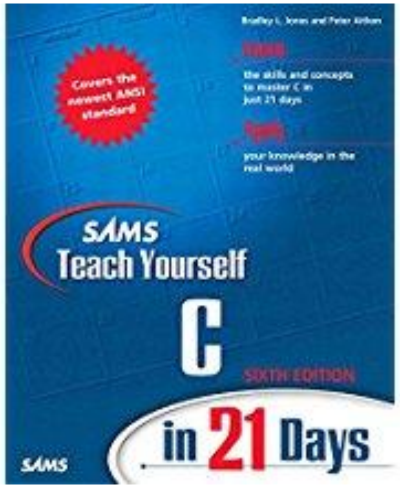
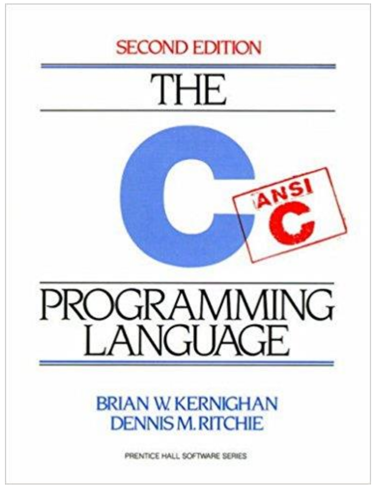
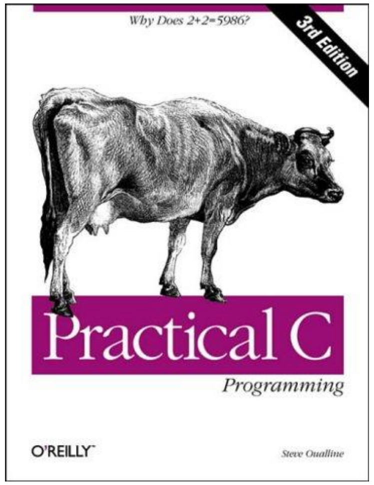
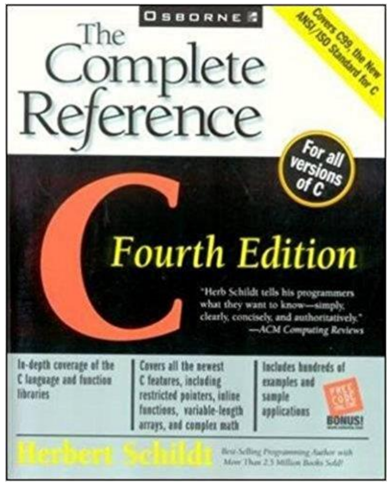
We have covered how data is stored on a computer and how it is represented in C.

You have learnt about functions, operators, both logical and relational and program control.

Finally we covered the basics of input and output.

You should now be in a position to write your own C program.

Further reading

http://www.learn-c.org/

https://www.cprogramming.com/tutorial/c-tutorial.html

https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

A deeper dive into C programming

Wes Armour

Oxford e-Research Centre,
Department of Engineering Science

Learning Outcomes

In this lecture you will learn about:

- More about arrays.
- Multidimensional arrays.
- An introduction to pointers.
- Characters and strings.
- Variable scope.
- Advanced program control.
- How to work with files.
- Dynamic memory allocation.

UNIVERSITY OF OXFORD

Oxford e-Research Centre

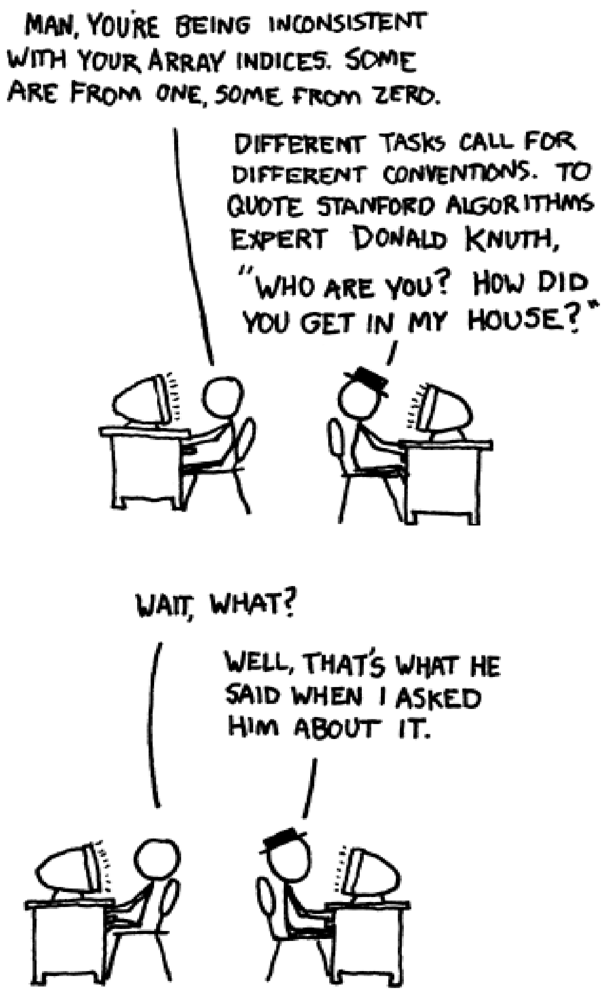
Using Arrays

In lecture two we presented the concept of arrays and how to use them. Now let’s look a them in some more detail. What characterises an array?

- An array is a collection of data storage locations.
- An array holds data all of the same type.
- Each storage location is called an *array element*.
- C arrays are indexed from 0 to n-1, where n is the number of elements in the array.
- Arrays can be initialised using braces:

```
int array[3] = {1,3,5};
```

As demonstrated in our second lecture arrays are a useful way to organise variables in your program.


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

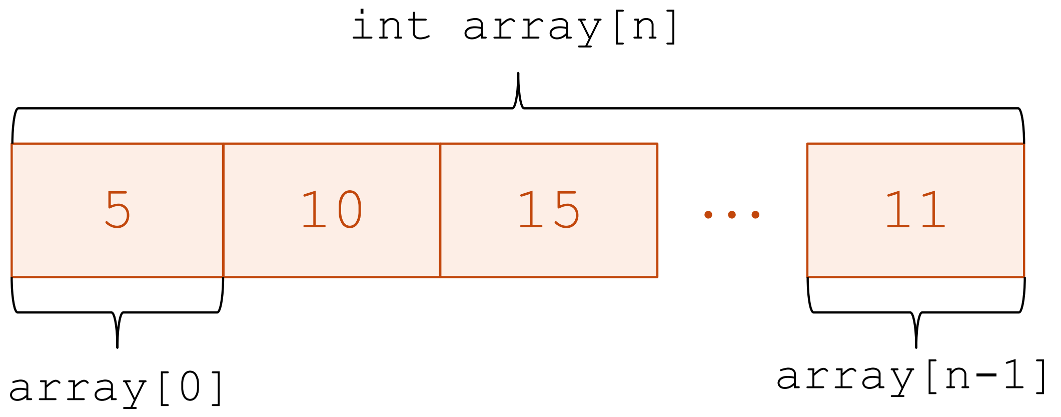
https://xkcd.com/163/

Using Arrays – single dimensional

In lecture two we used an array called radius to hold different values for the radius of a circle. This can be represented schematically below.

We have a *contiguous* collection of array elements, starting at zero, increasing to n-1, all holding a single value.

`array[0]` holds the integer value 5, `array[1]` holds the integer value 10, up to `array[n-1]` which holds the integer value 11.



Using Arrays – multidimensional

We can see that arrays can be very useful in helping us create compact and easy to read code. But what if we want to store something that has more than a single dimension?

Fortunately C has the concept of multidimensional arrays. Using these we can store an entity that has any dimension.

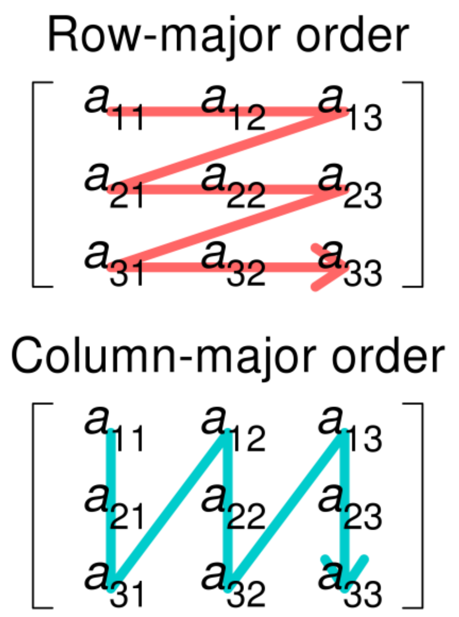
Lets consider the 3x3 identity matrix (right). We can store this as a 2D array which we declare as:

```
int identity[3][3]
```

In C this would tell the compiler to create a linear contiguous area in memory to hold 3x3 = 9 ints.

C uses **row-major ordering**. What does this mean?

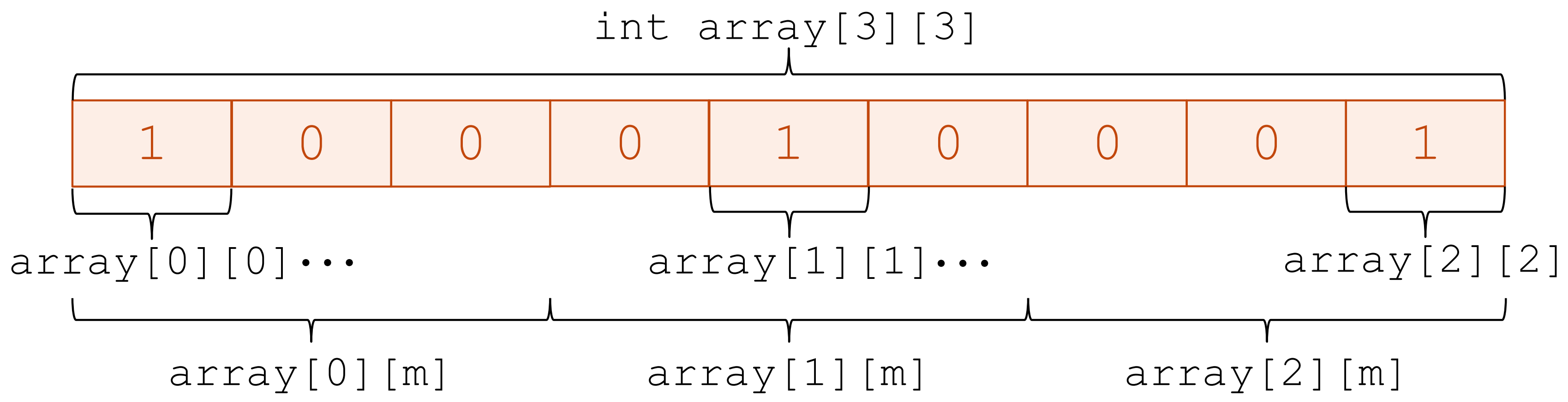
$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$


Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}
Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Using Arrays – multidimensional

Row-major ordering means that consecutive elements within a row reside next to each other in memory. Rows are then stored (in order) in memory consecutively.

Lets return to our example of the 3x3 identity matrix (below). We see the first element of the first row is stored first, followed by the second element of the first row, and so on. Then the first element of the second row is stored, followed by the second element of the second row etc.



Using Arrays – initialising multidimensional arrays

How do we initialise these arrays in our code?

Previously we showed that a one dimensional array can be initialised as follows:

```
int array[3] = {1,3,5};
```

We can initialise a 2D array in the following way:

```
int identity[3][3] = {{1,0,0},
                      {0,1,0},
                      {0,0,1}};
```

This can be generalised. The example below is the initialisation for a 3D array which has dimensions p x q x r (I've omitted actual values in an attempt to make this clearer).

```
int array[p][q][r] = {{{},{},{}},
                      {{},{},{}},
                      {{},{},{}}};
```

Using Arrays – using multidimensional arrays

Finally how do we used these arrays in our code? The code on the right gives an example of how we initialise a matrix with random numbers.

To do this we use the random number generator called `rand()`. This is a function that returns a random integer between 0 and RAND_MAX (where RAND_MAX is at least 32767). It has the function prototype:

```
int rand(void);
```

which is included in `stdlib.h`

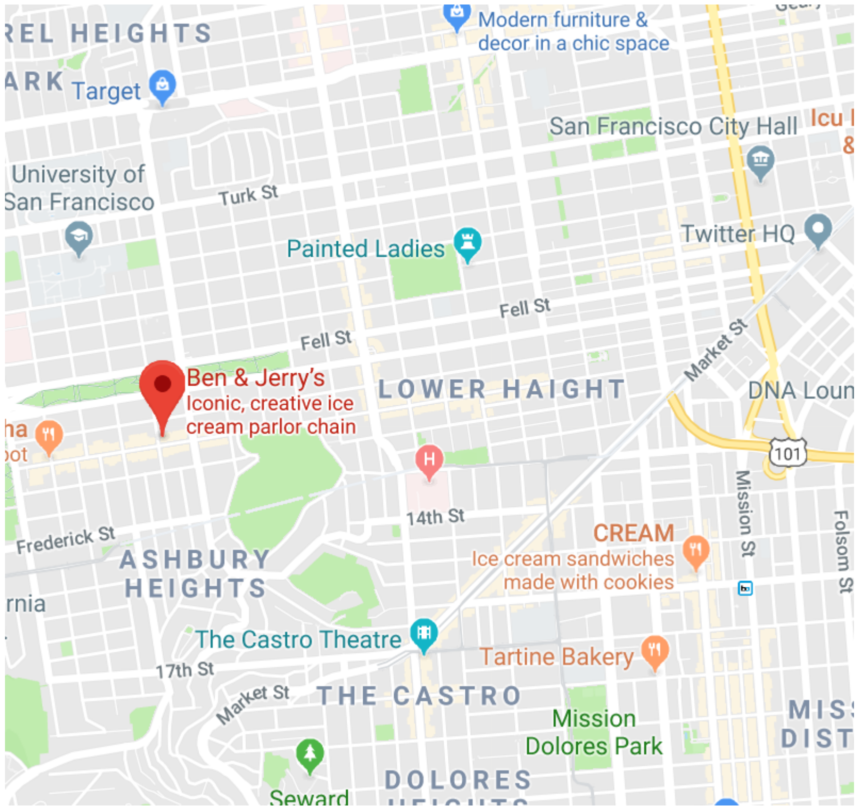
```
#include <stdio.h>
#include <stdlib.h>

int main() {

    const int els_in_row = 3;
    const int els_in_col = 3;

    int random[els_in_col][els_in_row];

    for(int col=0; col<els_in_col; col++) {
        for(int row=0; row<els_in_row; row++) {
            random[col][row]=rand();
        }
    }

    printf("\n\n");
    for(int col=0; col<els_in_col; col++) {
        for(int row=0; row<els_in_row; row++) {
            printf("%d\t\t", random[col][row]);
        }
        printf("\n");
    }
    return(0);
}
```

Understanding pointers - memory locations

Our previous example of arrays depicted a computers memory (RAM) as a sequence of linear, contiguous storage elements. We looked at how data items are stored in each element.

But how does a computer know in which element the data it wants to access is stored? For example how does it know where `identity[0][0]` is located?

This problem is overcome just as it would be in the real world. Each memory location is given a unique address. This is a simplified view of the actual mechanics of how memory is addressed, however it is sufficient for our needs.

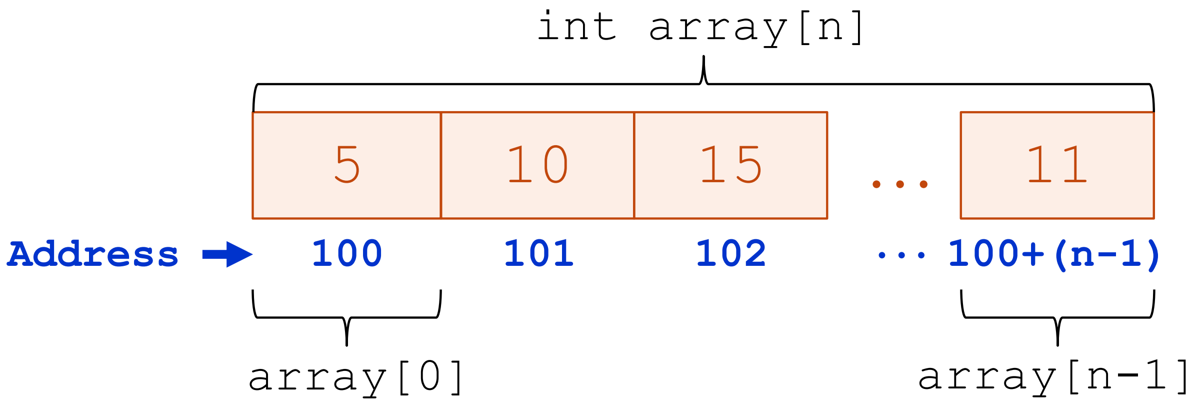


Understanding pointers – addressing memory

Lets go back to our previous description of an integer array.

But now lets add an address. I've chosen to add an integer address where the beginning element our `array` (which is `array[0]`) has address **`100`** (again this is simplified, but its sufficient to understand pointers).

Schematically this is represented on the right.



Understanding pointers – creating a pointer

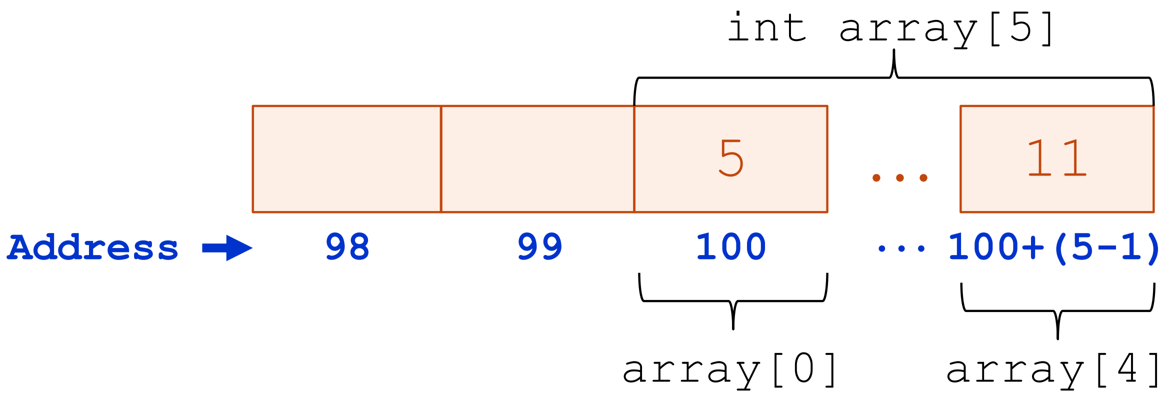
The next thing to note is that each address is a number and so can be treated like any other number in C.

If we know the address of `array[0]` then we could create another variable to store this address.

Lets work through the process for doing this over the next few slides.

Understanding pointers – creating a pointer

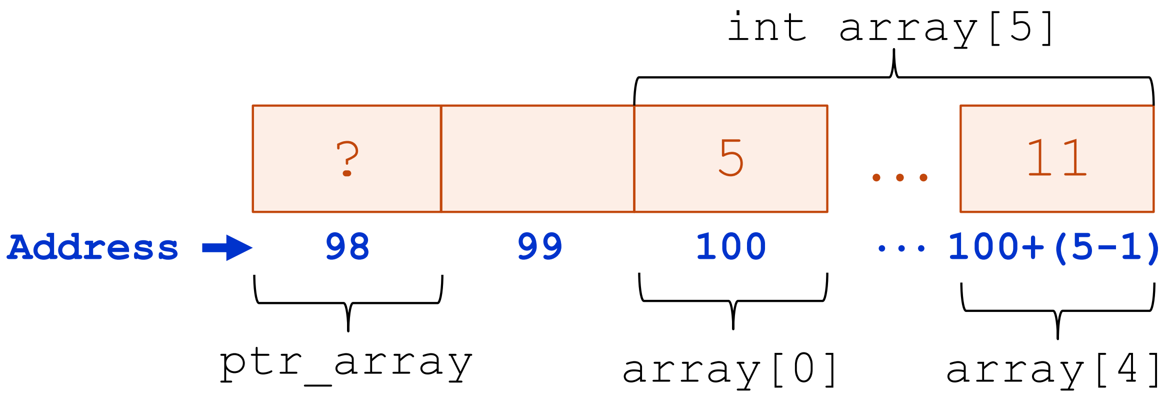
As before we declare our integer array, let's declare space for 5 integers (so n=5)



Understanding pointers – creating a pointer

Next we declare a variable (called `ptr_array`) that happens to live at address 98.

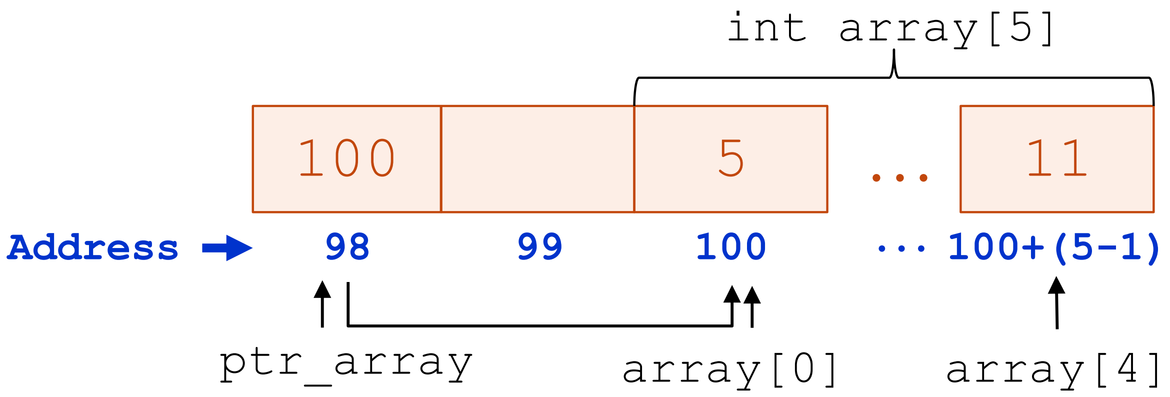
At this point it is uninitialized, so its value is undetermined.



Understanding pointers – creating a pointer

Now we **store the address** of `array[0]` in the variable `ptr_array`

Because `ptr_array` contains the address of `array[0]` it points to where `array` is stored in memory.



Hence `ptr_array` is a pointer to `array`

Understanding pointers – using pointers

To work with pointers we need to know about two operators. These are:

The indirection operator `*`

The address-of operator `&`

To understand these operators lets return to our simple example of calculating the area of a circle.

We declare a pointer to type float by:

```
float *ptr_radius;
```

The indirection operator tells the compiler that `ptr_radius` is a pointer to type float and not a variable of type float.

```
#include <stdio.h>

#define PI 3.14

int main() {

    float radius = 10.0;
    float area;

    float *ptr_radius;

    area = PI * radius * radius;

    return(0);

}
```

Understanding pointers – using pointers

To initialise the pointer (set it to point to something) we use the address-of operator:

```
    ptr_radius = &radius;
```

This tells the compiler to take the address of the variable `radius` and store it in the pointer `ptr_radius`

The code on the right shows how this works and tests the results of using a pointer.

```
#include <stdio.h>

#define PI 3.14

int main() {

    float radius = 10.0;
    float *ptr_radius;
    float area;

    area = PI * radius * radius;
    printf("\nArea:\t%f", area);

    area = 0;
    ptr_radius = &radius;
    area = PI * (*ptr_radius) * (*ptr_radius);
    printf("\nArea with pointer:\t%f", area);

    return(0);
}
```

Understanding pointers – using pointers

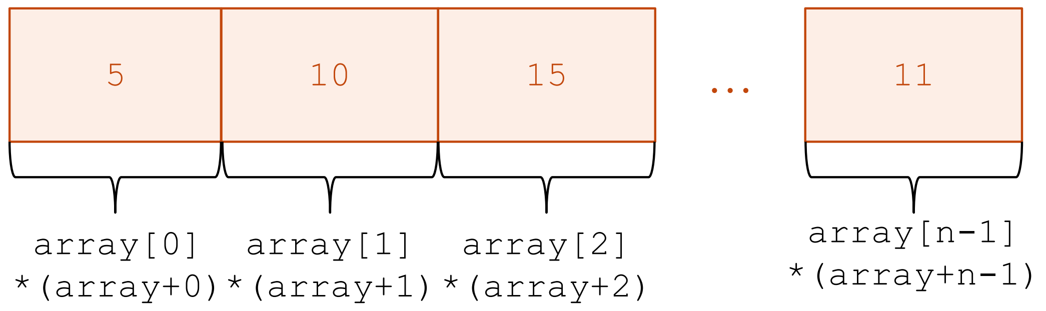
We can use arithmetic on pointers, just like we can any other numbers. A schematic demonstrating this is given below.

We also have increment: `ptr_radius++;`

and decrement operators: `ptr_radius--;`

Finally we can pass pointers as arguments to functions:

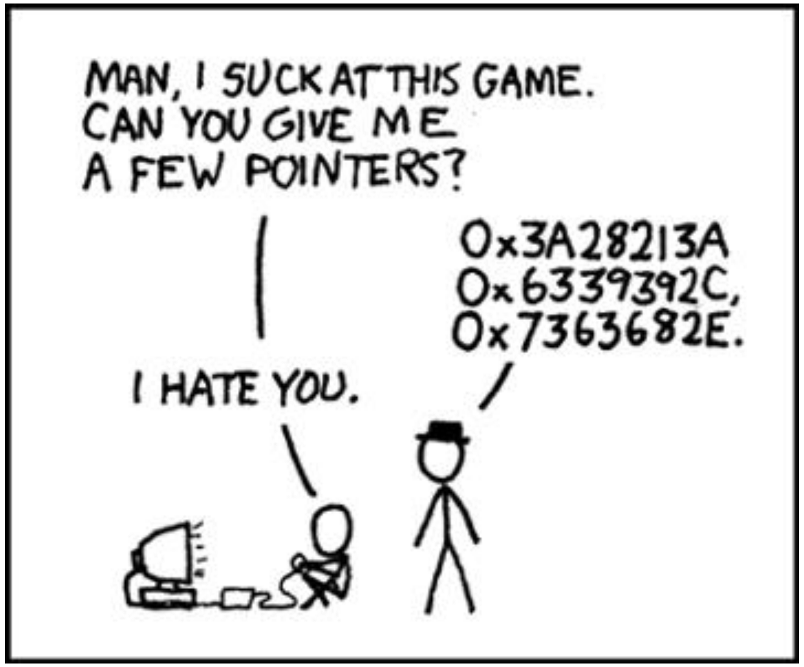
```
float area_of_circle(float *ptr_radius);
```



Understanding pointers

Pointers are (arguably) the most difficult concept to understand in C. However they are a powerful tool that can be used to write versatile and concise code. They also provide a flexible method for data manipulation.

In “Practical examples using the C programming language” we will look at the uses of pointers in more detail.



https://xkcd.com/138/

Break

Characters and Strings

C uses the char variable to store characters and strings.

C uses ASCII encoding to turn integer numbers into characters. An example of this is given on the right.

C decides whether a char holds a character or a number depending on the context of its use.

```
#include <stdio.h>

int main() {

    char one = 70;
    char two = ‘q’;

    printf(“\nOne as a character:\t%c”, one);
    printf(“\nOne as a number:\t%d”, one);

    printf(“\nTwo as a character:\t%c”, two);
    printf(“\nTwo as a number:\t%d”, two);

    return(0);
}
```

https://en.wikipedia.org/wiki/ASCII

Characters and Strings

```
[wa78@asteria0 ~]$ 
```

Characters and Strings

C uses arrays of char variables to store strings.

Stings are terminated with the null character which is represented by \0

So a string that has seven characters needs an array of eight elements to store it.

The code on the right gives an example of this and demonstrates two ways to initialise a character array with a string.

Recall the conversion specifier for a string is %s

```
#include <stdio.h>

int main() {

    char one[5] = {'H','a','r','d'};
    char two[5] = "Easy";

    printf("\nString one:\t%s", one);
    printf("\nString two:\t%s", two);

    return(0);
}
```

Characters and Strings

You can also allocate storage space for your string at compile time. To do this use one of the two ways demonstrated in the code on the right.

```
#include <stdio.h>

int main() {

    char one[] = {‘H’,’a’,’r’,’d’};
    char *two  = “Easy”;

    printf(“\nString one:\t%s”, one);
    printf(“\nString two:\t%s”, two);

    return(0);
}
```

Characters and Strings

For a user to interact with your program they need to be able to pass it input. C has two methods to read strings from the keyboard.

The first is the gets() function. This simply reads all input to the keyboard until a user presses the Enter key.

The second is the scanf() function, this requires the programmer to specify the format of the input using conversion specifiers.

Both methods are demonstrated in the code on the right.

```
#include <stdio.h>

int main() {

    char one[256];
    char two[256];
    char three[256];

    int count;

    printf("\nType some text and press Enter:\n");
    gets(one);
    printf("\nYou typed:\t%s", one);

    printf("\nType two words and press Enter:\n");
    count = scanf("%s %s", &two, &three);
    printf("\nYou entered %d words.", count);
    printf("\nYour words are: %s and %s", two, three);

    return(0);
}
```

Variable scope – Global variables

Variable scope refers to the extent to which different parts of your C program can "see" a variable that you declare.

The concept of scope allows a programmer to truly separate out (structure) their code into independent self contained routines or functions.

Doing this helps reduce bugs in code and makes for more reusable code. For example if a variable can only be seen by the function that is operating on it, another function cannot mistakenly corrupt its value.

In some instances it is desirable to share a variable amongst the whole code. This can be done with the `extern` keyword.

```
#include <stdio.h>

void print_number(void);

float one = 3.0;

int main() {

    extern float one;

    print_number(void);

    return(0);
}

void print_number(void) {

    extern float one;

    printf("\nYour number is:\t%f\n", one);
}
```

Variable scope – Local variables

External variables are sometimes called global variables. Their scope is the whole program, so `main()` and any other `functions()` that you define.

This is opposite to *local variables*. A local variable is defined within a function. As such its scope is within the function (remember `main()` is a function and so we can have local variables in `main()`).

Local variables are *automatic*, meaning they are created when the function is called and destroyed when it exits. So an automatic variable doesn't retain its value in between function calls.

To remember the value of a variable between function calls we can use the *static* keyword.

```
#include <stdio.h>

void print_number(int x);

int main() {

    for(int x=0; x<3; x++) {
        print_number(x);
    }
    return(0);
}

void print_number(int x) {

    static int y = 0;
    printf("\nx,y are:\t%d %d\n", x, y);
    y--;
}
```

Variable scope – Local variables

```
[wa78@asteria0 ~]$
```

Advanced program control

C provides some additional tools for advanced program control.

Three of the most useful are:

- `break`
- `continue`
- `switch`

Both `break` and `continue` provide additional control within loops. They are used within the body of a `for()` or `while()` loop, additionally `break` can be used in a switch statement.

The `switch` statement takes an argument and then executes code based on this.

```
#include <stdio.h>

void print_number(int x);

int main() {

    for(int x=0; x<5; x++) {
        print_number(x);
        if(x == 2) break;
    }

    int x = 0;
    while(1) {
        if(x == 3) {
            break;
        }
        print_number(x);
        x++;
    }
    return(0);
}

void print_number(int x) {
    static int y = 0;
    printf("\nx,y are:\t%d %df\n", x, y);
    y--;
}
```

Advanced program control

The switch statement is useful when you want to compare a value against a list of known values. An example might be allowable responses for input. More general comparisons are done by `if` as we saw previously.

An example of how the `switch` statement can be used is given on the right.

```
#include <stdio.h>

void print_number(int x);

int main() {

    int choice;
    printf("\nEnter a choice: 1,2 or 3 to exit: ");
    scanf("%d", &choice);

    switch(choice) {
        case 1:
            printf("You entered 1");
            break;
        case 2:
            printf("You entered 2");
            break;
        case 3:
            printf("You entered 3. Exiting.");
            exit(0);
    }
    return(0);
}
```

Using files

C provides a number of different tools for interacting with files stored on your computer.

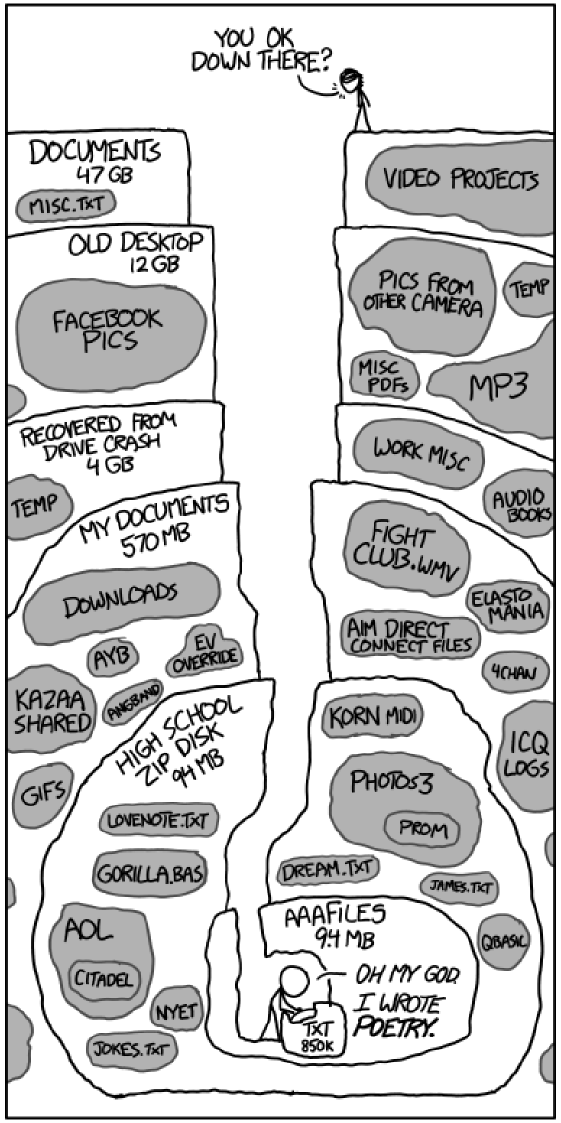
In this lecture we'll cover four basic functions for this.

fopen()

fclose()

fprintf()

fscanf()



https://xkcd.com/1360/

Using files – fopen and fclose

The code on the right declares a pointer to type FILE. Then two char arrays are declared to hold a file name and a mode.

The file name is the name of the file you want to use and the path to it.

We use `gets()` to read strings from the keyboard and store these in our char arrays.

We then use `fopen()` to try and open the requested file. If the file opens successfully then the pointer fp is initialised. If not the pointer is set to NULL and we print an error.

Finally we close the file using the `fclose()` statement.

```
#include <stdio.h>

int main() {

    FILE *fp;
    char filename[200], mode[4];

    printf("\nEnter your file: ");
    gets(filename);
    printf("\nEnter a file mode: ");
    gets(mode);

    if((fp=fopen(filename, mode)) != NULL) {
        printf("\nOpened %s in mode %s", filename, mode);
    } else {
        printf("\nERROR: File not recognised");
    }
    fclose(fp);
    return(0);
}
```

Using files – fopen and fclose

In the previous slide we introduce the concept of opening a file in different *modes.*

The table on the right outlines different modes and the and the associated return value of `fopen()`

Mode	Meaning	Return value from fopen if the file:	
		Exists	**Doesn't exist**
r	Reading	–	NULL
w	Writing	Overwrite if file exists	Create new file
a	Append	New data is appended at the end of file	Create new file
r+	Reading + Writing	New data is written at the beginning of the file overwriting existing data	Create new file
w+	Reading + Writing	Overwrite if file exists	Create new file
a+	Reading + Appending	New data is appended at the end of file	Create new file

Using files – fprintf and fscanf

The code on the right gives examples of using the `fprintf()` and `fscanf()` functions (it’s a bit squashed).

We open a file as before (note the bad coding practice, the code fails silently if `fopen()` fails).

We use `fprintf()` to print 10 numbers and their squares to a file. Note how `fprintf()` works like `printf()`

We close the file and reopen it.

We then use `fscanf()` to read in our outputted numbers.

Finally we perform a difference of these and print out to screen.

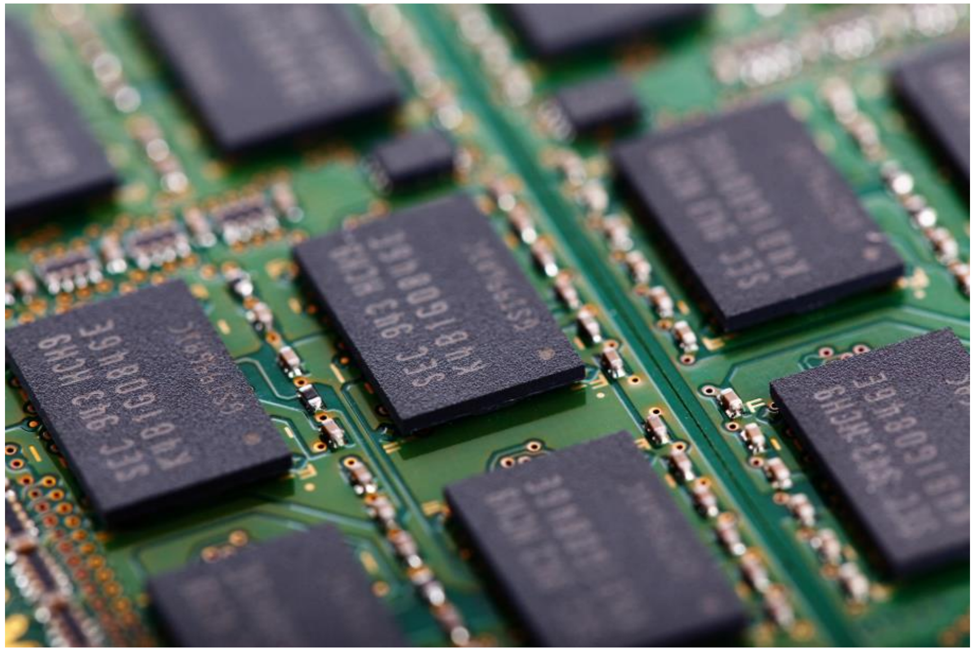
```
#include <stdio.h>

int main() {

    FILE *fp;
    char filename[200], mode[4];
    int index[10], square[10];

    printf("\nEnter your file: "), gets(filename);
    printf("\nEnter a file mode: "), gets(mode);
    if((fp=fopen(filename, mode)) == NULL) exit(1);

    for(int i=0; i<10; i++) {
        fprintf(fp, "%d %d", i, i*i);
    }

    fclose(fp);
    fp=fopen(filename, mode);

    for(int i=0; i<10; i++) {
        fscanf(fp, "%d %d", &index[i], &square[i]);
    }
    for(int i=0; i<10; i++) {
        fprintf(fp, "%d %d", i-index[i],(i*i)-squared[i]);
    }
    return(0);
}
```

Working with memory

Up until now all of our example codes have allocated *static memory.*
By this we mean that when we write our code we declare how much memory we need for particular variables or arrays.

However there might be instances where we don’t know how much memory we need until we run the code. For example we might want to read a file into our code that is updated on a daily basis, the files length might be different on different days.

C provides a process for allocating memory at *runtime*. This is called *dynamic memory allocation*.



https://en.wikipedia.org/wiki/C_dynamic_memory_allocation

Working with memory

Lets return to our example code from Lecture two. We can modify this code so that is asks the user how many areas they want to calculate and then dynamically allocates memory for them.

We start by declaring a pointer:

```
float *radius;
```

We then use `scanf()` to get the number of circles the user wants to work with.

We then use `malloc()` to allocate the memory that we need.

After this we use `scanf()` to get the radii from the user.

This time we accumulate directly to `total_area`

Finally we `free()` our allocated memory.

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.14

float area_of_circle(float radius);

int main() {
    int i=0, number_of_circles=0;
    float *radius;
    float total_area=0;

    printf("\nEnter the number of circles to calculate:\t");
    scanf("%d", &number_of_circles);

    radius=(float *)malloc(number_of_circles*sizeof(float));

    printf("\nEnter the radii:\n");
    for(i=0; i<number_of_circles; i++) scanf("%f", &radius[i]);

    i=0;
    while(i<number_of_circles) {
        total_area += area_of_circle(radius[i]);
        i++;
    }
    printf("\nTotal area is:\t%f\n", total_area);

    free(radius);
    return(0);
}

float area_of_circle(float radius) {
    float area = PI * radius * radius;
    return area;
}
```

Working with memory – malloc()

Lets take a closer look at `malloc()`

Both `malloc()` and `free()` are defined in `stdlib.h`, so this must be included into your code to use them.

The function prototype for `malloc()` is:

```
void *malloc(size_t num);
```

`size_t` is an unsigned integral type and is used to represent the size of an object in bytes. The return type of the `sizeof()` operator is `size_t`

`malloc()` will return NULL if `num` bytes cannot be allocated (for example the computer doesn't have enough memory space left)

$$\overbrace{\text{m}}^{\text{memory}}\underbrace{\text{alloc(size)}}_{\text{allocation}}$$

Working with memory – free()

Once we've finished working with the memory that we've allocated using `malloc()` we should free it so that it can be used again.

This is done using the `free()` function. Its function prototype is:

```
void free(void *ptr);
```

Calling `free()` releases the memory that is pointed to by `ptr`

Working with memory – multidimensional arrays

Finally lets return to our identity matrix.

`int identity[3][3]`

How can we allocate memory for this using `malloc()`?

The code snippet on the right shows how to do this and then free the allocated memory using `free()`

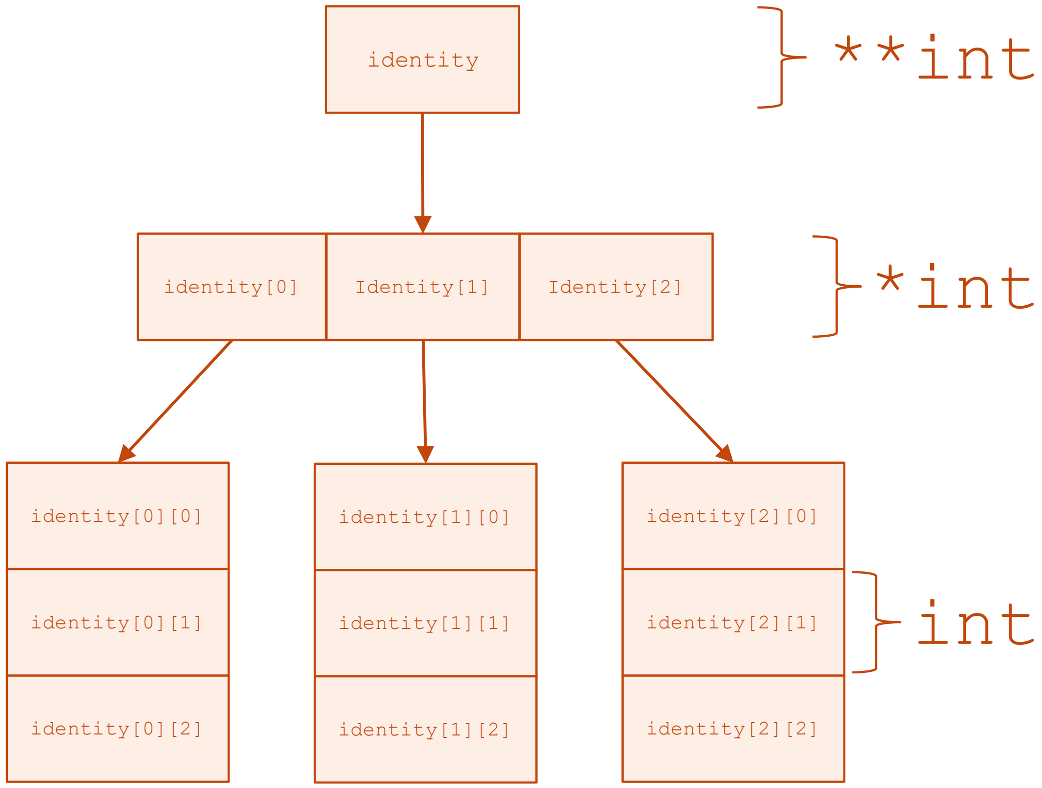
```
int ** identity;

int num_rows = 3;
int num_cols = 3;

// To allocate

identity = (int **)malloc(num_rows * sizeof(int*));
for(int i=0; i<num_rows; i++) {
    identity[i] = (int *)malloc(num_cols * sizeof(int));
}

// To free

for(int i=0; i<num_rows; i++) {
    free(identity[i]);
}
free(identity);
```

Working with memory – multidimensional arrays

Schematically this can be represented by the diagram on the right.

We begin by allocating a double pointer to type `int**` This in turn points to an array of pointers of type `int*`

We then point each of these at a single dimensional array of type `int`.



What have we learnt?

In this lecture you have learnt about some of the advanced features of the C programing language.

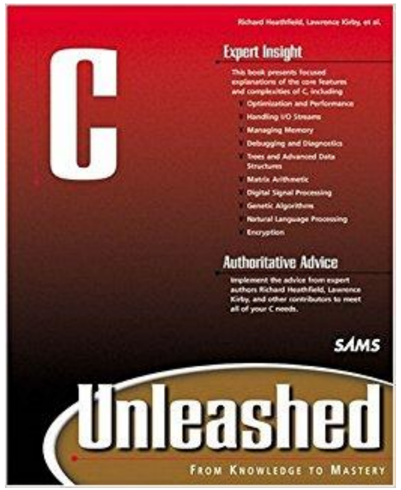
We have covered multidimensional arrays, pointers, characters and strings.

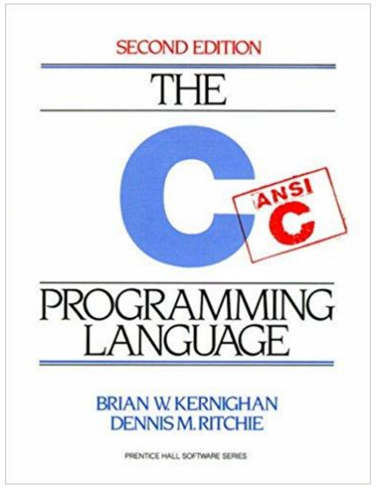
You have learnt about variable scope, some of the functions that provide advanced program control and how to work with files.

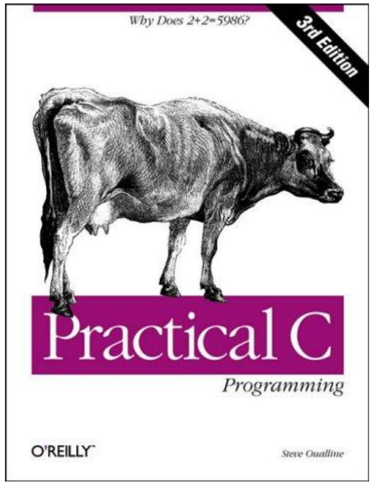
Finally we have covered the basics of dynamic memory allocation.

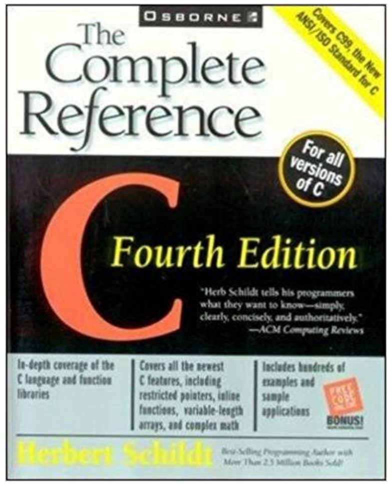
You should now be in a position to write your own advanced C program.

Further reading


C
Unleashed
SAMS
Richard Heathfield, Lawrence Kirby, et al.
From Knowledge to Mastery







http://www.learn-c.org/

https://www.cprogramming.com/tutorial/c-tutorial.html

https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html