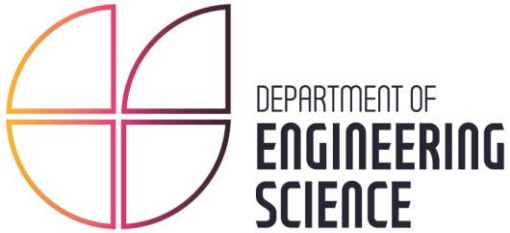


An Introduction to HPC and Scientific Computing

Lecture two: Introduction to the C programming language

Wes Armour

Oxford e-Research Centre,
Department of Engineering Science

Overview

In this lecture you will learn about:

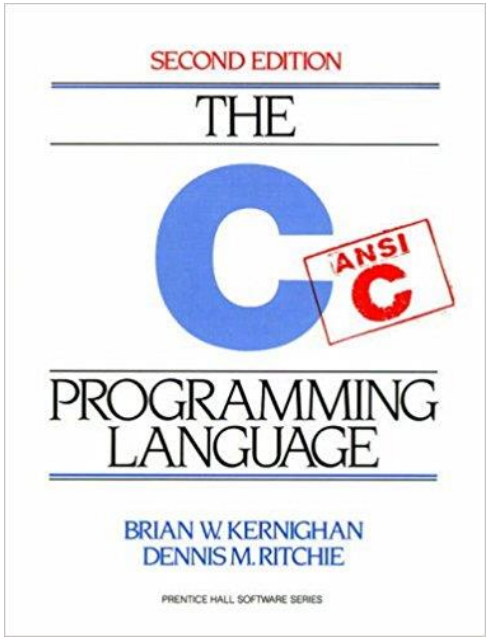
- High level computer languages.
- The basic components of a C computer program.
- How data is stored on a computer.
- The difference between statements and expressions.
- What operators and functions are.
- How to control basic input and output.
- Finally how to write a basic C program.

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 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.

```
int square(int num) {

    return num * num;

}
```

HLL

```
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
```

Assembly Code

```
01110000011010001010100100100101001
00011111111111101010111100111001010
10101010011111111111101010100101010
10101010101010101001010101010100101
```

Machine Code

https://godbolt.org/

UNIVERSITY OF OXFORD

Oxford e-Research Centre

The components of a C program

All C programs have some common elements.

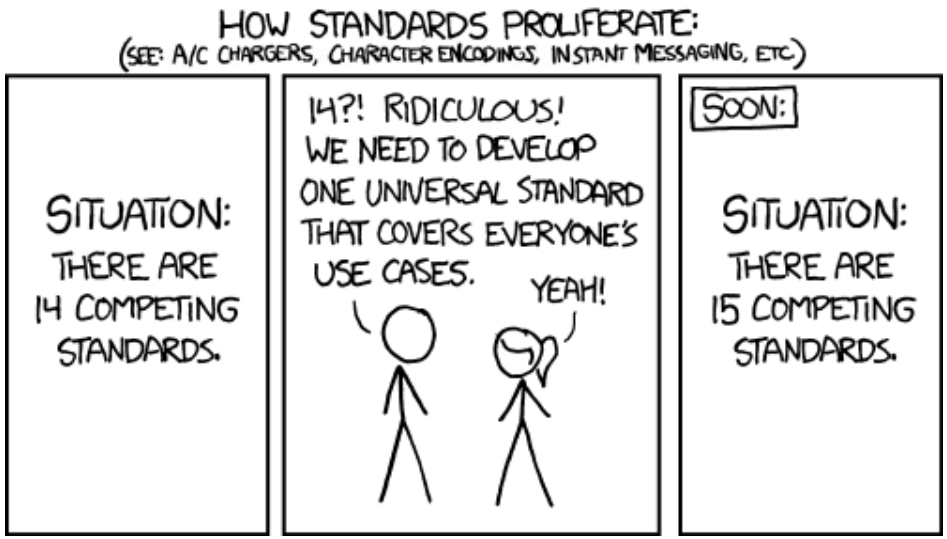
- The `#include` directive tells the compiler to include other files stored on your HDD into your C program. These files will include information that does not change between programs that your program can use. On the right we include the standard input and output library.
- The `main()` function is 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 is 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 that are contained in the C standard library will save you time, effort and make your code more understandable to others.



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

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 bit 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 your program execution down.

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 you 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

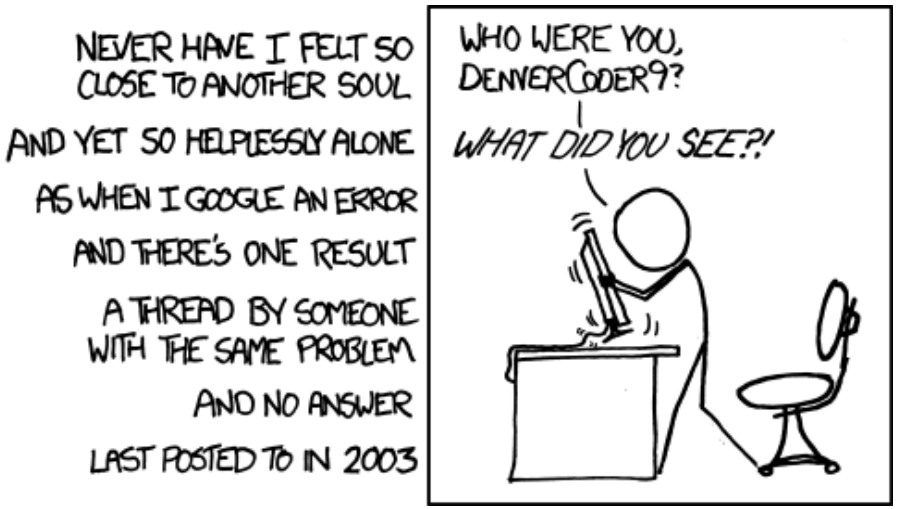
```
area =
       PI *
          radius
                    * radius
             ;
```

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 */



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.

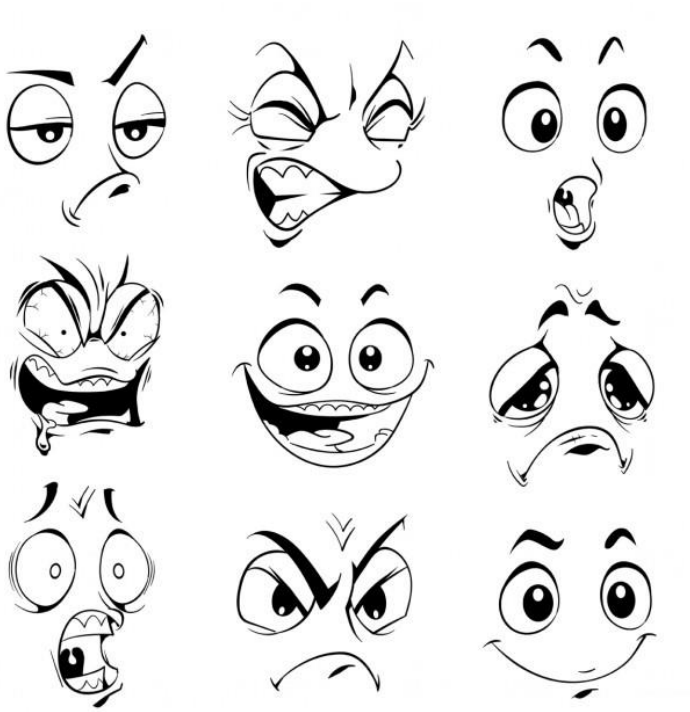
```
/* This is a C program that
   calculates the area of a circle.
   Written by Wes
   wes.armour@eng.ox.ac.uk
   06/05/18
*/

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

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;
}
```

UNIVERSITY OF OXFORD

Oxford e-Research Centre

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;
}
```

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.

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.

An example on the left checks to see if we have a radius less than our 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 circumfernece;

    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 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;

    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;
}
```

UNIVERSITY OF OXFORD
Oxford e-Research Centre

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;
```

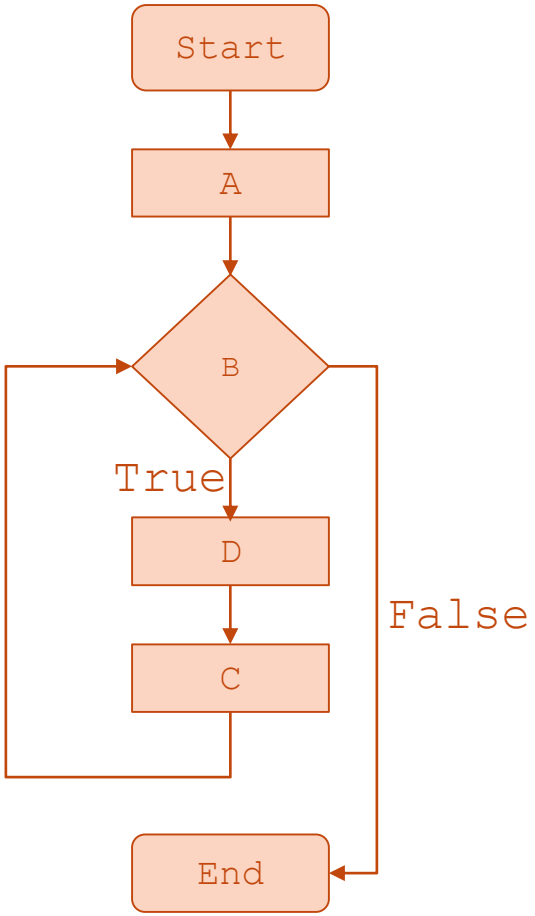
The initial starting point A is evaluated.

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.

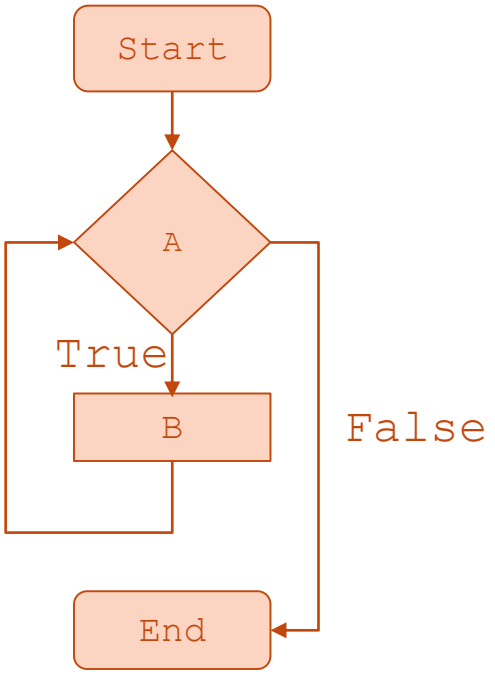
Consider the following while loop:

```
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)!

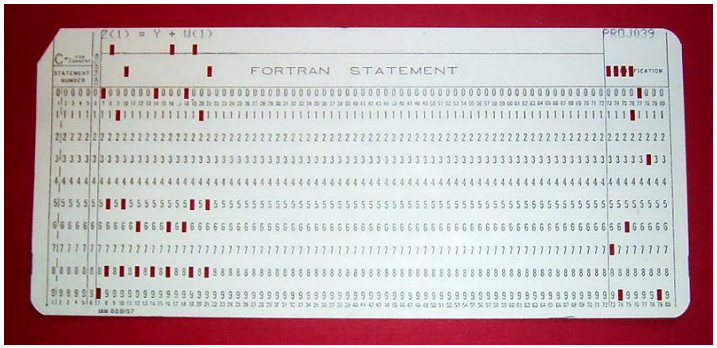
Fortunately the C language has many functions to help with input and output.

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;
    float radius[3];
    float area[3];
    float total_area;

    scanf("%f %f %f", &radius[0], &radius[1], &radius[2])

    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

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.

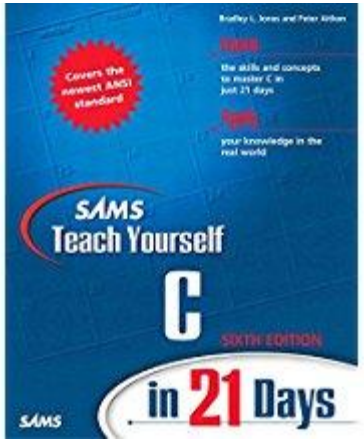
We have covered how data is stored on a computer and how it is represented in C.

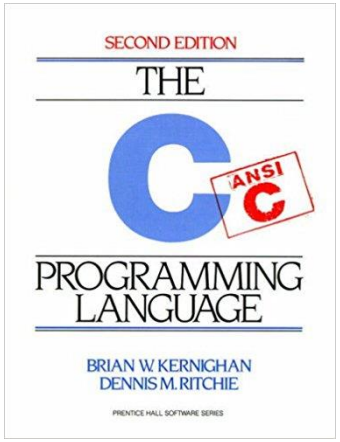
You have learnt about functions, operators, both logical and relational and program control.

Finally we covered the basics of input and output.

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

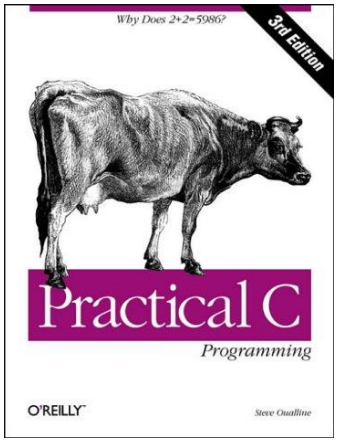
Further reading

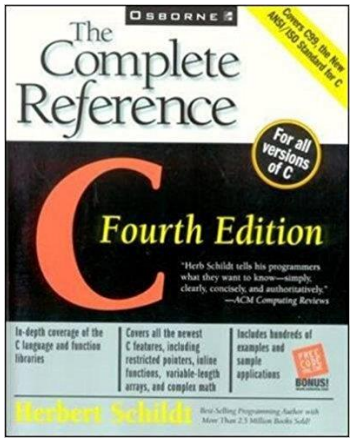




http://www.learn-c.org/

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





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

UNIVERSITY OF OXFORD

Oxford e-Research Centre

In the next lecture…

In the next lecture you will learn about the Linux operating system and how to use it. You'll get to know compilers, what they do and how to use them. You'll also learn about build systems, what they are, how to use them and how they can make the life of a programmer easier.