UQC146S1
Introductory Image Processing in C

Ian Johnson

Room 3P16
Telephone: extension 3167
Email: Ian.Johnson@uwe.ac.uk
http://www.csm.uwe.ac.uk/~irjohnson/uqc146s1.html
What is this module about?

The C Programming Language

Basic Image Processing

Books?

"C Programming", Tony Royce, Macmillan 1996 is strongly recommended.

CRTS Students will have already had this book recommended!

You will be expected to read this!
We only have a limited time to introduce C before moving on to image processing.

Other books:
"Simple C", Jim McGregor et. al., Addison-Wesley, 1998

Assessment?

One large assignment, due end of term 2 (75%)

- You will need to start early!
- Passing this module without passing the assignment is virtually impossible.

One examination (25%)

- Plenty of time allowed
- Section A – short answers to small questions (70%)
- Section B – one question from a choice (30%)
Tools & Environment

Working in the Solaris laboratories
   Solaris is Sun Microsystems proprietary UNIX

Sun workstations

Sun workshop C IDE
   (see worksheet 1)

   Based on the editor of your choice, Emacs, Xemacs or Vi
   (emacs reference cards available from the helpdesk)

Pbmplus, Xv, & Netscape
   (see worksheet 3)

Pbmplus, Xv, Netscape, the GNU C compiler & debugger (gdb), & GNU
emacs and Xemacs are all freely available on virtually every flavour of
linux.

Starting at C

C is not an object-orientated programming language

C is not C++ or Java (although both are based on C)

We will not be covering the language in depth, but rather a useful subset.
To become proficient you will need to work your way through Royce or
an alternative book.

In the early weeks, complete the random number exercises and/or the
exercises from Royce.

THE ONLY WAY TO LEARN TO PROGRAM IS TO PROGRAM!
To get the most from C you will need to know what functions are available in the standard libraries.

**Starting at the beginning ...**

... (or rather *main*)

All C programs consist of functions. Execution of a C program begins with the function `main()`.

It therefore follows that all C programs must have at least one function, and one & only one called `main()`.

A function is declared in C as:

```
Return_Type Function_Name(Arguments) {
    /* function contents */
}
```

Functions can call other functions and make use of their return value, or pass then data to work with as arguments.

This is how we can reuse other people's code, and modularise our own.

All C compilers depend on libraries of standard functions to be useful, since the language itself is very small.

All libraries have an associated header file which contains prototypes for the functions, (so the compiler can check that you're using them correctly), together with data structure definitions, and other useful definitions.

We can declare both the Return_Type & Arguments to be *void*

A Special case where we don't wish to pass arguments or return a value.
Comments

/* this is a comment */

Comments should add meaning to a program, and aid understanding.

day = day + 3 % 7 /* add 3 to day modulo 7 */

Does not. If you can read C, the comment is pointless, if you can't – why are you trying to edit this program?

Try to explain how and why you are doing something.

day = day + 3 % 7 /* skip the weekend */

Beware:

// This is a C++ comment **not** a C comment

Many C compilers today also compile C++. Some determine which is which by file extension (.C, or .cpp for C++), others guess. Some accept C++ comments, others do not.

DO NOT USE C++ COMMENTS IN C
Types

All functions return a value of a specified type. All variables have a specific type. The type of a variable tells the compiler how much memory to allocate to hold the variable, and what it will be used for. The basic types follow, more later!

- **int**  An integer value. int’s in C are of the most convenient size for the architecture, typically today 32 bit.

- **char**  a byte (8 bit) sized integer, convenient for holding ASCII characters (which require 7 bits).

- **float**  a single precision floating point number.

- **double**  a double precision floating point number.

we can also have “short int” or short and “long int” or long

Unlike some languages (e.g. occam2) which define the exact size of a variable, C does not.

Instead:

16bit <= short <= int <= long
long >= 32bit

long double is also allowed

All integer types can be **unsigned**

Unsigned char 0 … 255
char -128 … 127

We also have register, static & volatile modifiers which we’ll discuss later.

All minimum & maximums are defined in the standard header file <limits.h>

e.g. INT_MAX
Standard header (or include files)

C depends on libraries to be useful. The language itself is very small.

All libraries have an associated header file, which prototypes functions (so that usage can be checked), contains data structure definitions, and other useful definitions (e.g. the maximum size of data types as discussed above).

the notation:

<headerfile.h>

says include headerfile.h from the system location of header files. On a unix system, this is often /usr/include by may not be.

Where are include files kept for Visual Studio?

If you create your own header files then you can reference them as

“myheader.h” (if it is in the current directory)

or

"../include/ myheader.h"

"/home/staff/csm/irjohnso/project1/myheader.h"
Variables

variable names in C start with a letter, and can contain upper and lower case letters, digits and the underscore “_” character.

C is case sensitive, C and c are not the same!

To declare an integer variable:

```c
int my_integer_variable;
```

Convention in C is that lower case names are used for variables, all upper case names for defined constants:

```c
int my_integer_variable = INT_MAX;
```

Constants

Whilst C has a `const` type modifier, it is rarely used:

```c
const double pi = 31.4159;
```

far more frequently, C programmers will use the pre-processor directive:

```c
#define PI 3.14159
```

These behave differently. You can think of `#define` as copying the text 3.14159 into your program to replace every occurrence of PI.

Constants have a default type

1000 is an int
1000L (or l) is a long
1000U (or u) is an unsigned int

Integers can also be specified in octal (base 8) or hexadecimal (base 16). A leading 0 implies octal, a leading 0x implies hexadecimal.

3.14159 is a double
3.14159F (or f) is a float
doubles can also be represented in exponent form, e.g. 1e3 is 1000 stored
as a double.

Character constants are written as 'a'. We can also specify a character
escape as:
'\ooo' (3 octal digits)
or
'\xhh' (2 hex digits)

Remember, characters are treated as numbers!

```c
char x;
x = 48  x = '\x30'  x = 0x30  x = '0'
```
all give x the same value

We also have some predefined special characters:

```
\b  backspace   \a  alert (bell)
\n  newline   \r  carriage return
\t  tab       \v  vertical tab
\  \       \?  ?
\\  "    \f  formfeed
\'  '      \0  the null character
```
Strings

String constants such as

"Hello World"

are specified in double-quotes.

It is vitally important to remember that "a" and 'a' are NOT the same.

Strings are arrays of characters with an ASCII NULL (value 0) as the last character.

Characters are single byte values.

A first program

As is traditional hello world

```c
#include <stdio.h>  /* include standard C I/O header */

void main(void)   /* declare a function called main, */
{    /* taking no arguments returning no value */
    printf("Hello World\n");  /* use the library function
       printf to output our
       string */
    return;                    /* return from the function */
}
```

When we return from `main()`, we exit to the operating system.

When other functions return, control passes back to the calling function.

You should have already built this program using workshop.

Try compiling on the command line using:

`cc -o hello hello.c`
Operators

All operators have precedence, (that is some are evaluated before others) and associativity (which direction they are evaluated in).

\[ x = 3 \]

associates right to left, this should be commonsense, others might not be.

We’ll revisit this shortly.

Assignment $=$

Binary Arithmetic $+$ $-$ $*$ $/$ as expected

$\%$ modulo

Binary in this case means 2 operands!

Increment / Decrement ( $++$ $--$ )

$x--$ use x, then decrement

$--x$ decrement x then use

Bitwise:

& AND

| OR

^ XOR

<< left shift

>> right shift

~ one’s complement

The above operators can be combined with $=$

\[ \text{expressionA} \ \text{operator=} \ \text{expressionB} \]

means the same as:

\[ \text{expressionA} = (\text{expressionA}) \ \text{operator} \ (\text{expressionB}) \]

Be careful:
\begin{verbatim}
x = 3;
y = 4;
x *= y + 2;
\end{verbatim}

What is the new value of \( x \)?

\begin{verbatim}
relational: > >= < <= == !=
\end{verbatim}

**Boolean (or not) Conditions**

C takes an unusual approach to evaluating conditions.

Zero = False
Non-Zero = True

This allows side-effect laden conditions, NOT A GOOD IDEA.

**Precedence and Associativity of Operators in C**

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ) [ ] -&gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ ++ -- + - * &amp; (type) sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;= &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>^ &amp;</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
</tr>
</tbody>
</table>

Unary +, - and * have higher precedence than the binary forms.
taken from: Kernighan & Ritchie, p. 53

What is the value of: \( 3 * 4 + 1? \)
Advice: Bracket wildly, do not overcompound statements.

Use brackets to make your meaning clear and ensure the accuracy of your expressions.

Control Flow

Like most programming languages C has various constructs for control flow.

Selection

If (you can keep your head while all around…)

if (condition) statement;

if (condition) statement else statement;

BEWARE:

if (x=0)
    y=4;

is legal C, x will be assigned the value 0, the if will evaluate to false, and y will never be altered.

In C Boolean expressions are numerically evaluated

false = ZERO
true = NOT ZERO
Therefore:

```c
if (x-2) statement;
```

is legal. Subtract 2 from x and if the result is not zero perform statement.

```c
if ((x-2) != 0) statement;
```

says the same, but is more readable!

**Other selections**

**Switch**

```c
switch (expression)
{
    case const-expression: statements;
    default: statements;
}
```

Switch statements can be very useful, their design in C is a little tricky however.

Execution begins at the statement following the case that `expression` evaluates to and continues to the end of the switch. Normally you will want to place a `break` statement at the end of each case.

**The Ternary operator**

Beloved by those who like to write impenetrable C, it can be occasionally useful.

```c
expression1 ? expression2 : expression3
```

expression1 is evaluated, if true then expression2 if false expression3
Repetition

C provides a set of looping or repetition operators:

**While**

while (condition) statement;

do
   statement
while (condition);

While loops are only executed if condition is true, whilst do-while loops are executed at least once.

**For**

For loops in C can be simple counted loops but are far more powerful, and often abused!

for (init-expr; test-expr; repeat-expr)
   statement;

init-expr is performed at the start of the loop. The loop is repeated while test-expr is true, with repeat-expr being executed for each iteration.

Used sensibly:

for (i=0; i<10; i++)
   dosomething();

Used differently:

char x[10];
for (i=0; i < 10; x[i++]=0);