UNIVERSITY OF EXETER

School of Physics

Revsion of PHY2004

Comments and questions to John Rowe.

Types of variables

C has two main types of variables, **integers** and **floating-point**. The most common type of integer is called an **int**, although long and long long are available if a greater numeric range is required. For floating-point calculations the float type is suitable for low-precision calculations but **double**s are preferred where accuracy is required.

The computer stores the values of its variables in its memory

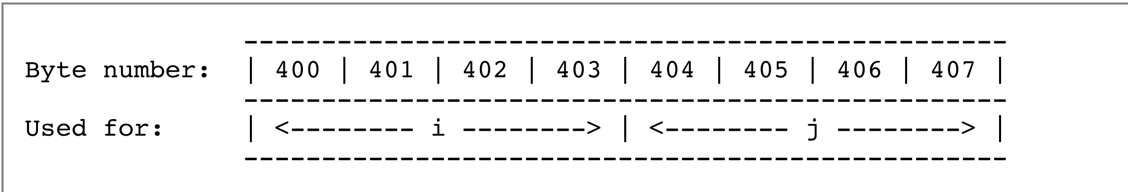
At the core of any computer are its processor unit (CPU) and its memory (RAM - for Random Access Memory). Memory is measured in bytes and we can think of it being arranged in a block with the first byte numbered one, the second two, etc..

Typically it takes four bytes (but may be as little as two bytes on some systems) to store an `int`, so when the computer encounters a line like:

> A separate standard covering computer hardware says that `float`s are four bytes long and `double`s eight. You are unlikely to encounter a C compiler that does not obey this.

```
int i, j;
```

it reserves two chunks of four bytes each to store the values of these two variables:



```
              -------------------------------------------------
Byte number:  | 400 | 401 | 402 | 403 | 404 | 405 | 406 | 407 |
              -------------------------------------------------
Used for:     | <-------- i --------> | <-------- j --------> |
              -------------------------------------------------
```

In this example and all the examples below we have assumed four byte ints and we have arbitrarily assumed that the compiler has decided to put i and j next to each other with i beginning in memory location 400.

Notice how, as far as the computer is concerned, references to variable names such as `i` and `j` are **just ways of referring to a particular part of the memory**.

Steps for the simple Assignment.

eg `j = i + 1;`

1. Read the values stored in the memory assigned to the variables on the right hand side of the statement into the central processor.
2. Perform the calculation, keeping the result in the processor.
3. Store this value in in the memory assigned to the item on the left hand side of the statement.

Short cuts

C programmers love to express things compactly. Some examples:

Long form	Short form
`y = 1;` `x = y;`	`x = y = 1;`
`x = x + 1;`	`++x;`
`x = x - 1;`	`--x;`
`x = x * y`	`x *= y;`
`x = x + y`	`x += y;`

We can use the `*=` type notation for any of `+, -, *, /, <<, >> or %` .

Experienced C programmers use the above notations all the time, both because they express a simple concept ("x equals y equals one", "multiply x by y") very simply and because these expressions arise so frequently

that you very soon get to recognise them. For this reason, we prefer you to use these shortcuts wherever they appear in a statement on their own.

When they appear in combination or as part of a more complicated statement you should feel free to use either form, depending on which you feel is clearer. For example, either of:

```
y *= x;
x = y;
```

or

```
x = y *= x;
```

is legal C, but I feel the two line version is clearer. **Use shortcuts to make things simpler and clearer, not more confusing.**

Integer arithmetic

The % operator

If `i` is a positive integer and `j` a strictly positive integer `i % j` gives the remainder:

```
7 % 2 == 1
9 % 5 == 4
6 % 3 == 0
```

It's best not used with negative numbers.

Integer division

Similarly, if `i` `j` are integers `i % j` gives an **integer** result, rounding towards zero:

```
7 / 2 == 3
9 / 5 == 1
6 / 3 == 2
```

Again, it's best not used with negative numbers.

```
( i / j ) * j + i % j == i
```

whether i and j are positive or negative.

Negative values

If do find the need to use integer arithmetic for negative values the rule for integer `j` and `k` is :

```
-j/k equals j/-k equals -(j/k)
-j % k equals j % -k equals -(j % k)
```

i.e. integer division always rounds towards zero.

Complex arithmetic

The original C standard did not include complex arithmetic, as it is mainly used by scientists and engineers, but it has been available since C99.

Complex arithmetic is enabled by:

```
#include <complex.h>
```

which enables the use of the `double complex` data type and the imaginary constant `I` (note capital).

Complex versions of most of the `math.h` functions are also inside `complex.h`, all with a **`c`** in front of the name such as `csqrt()`, `csin()`, `casin()`, `cexp()`, etc.

The functions **`creal()`**, **`cimag()`** give the real and imaginary parts respectively with `conj()`, `cabs()` and `carg()` also available.

Notice the difference between `sqrt(-1.0)` (bad) and `csqrt(-1.0)` (good).

Arrays

Arrays give us the ability to define a collection of variables all of the same type and to be able to refer to them all by the same name with a numerical index as in this rather silly example:

```
int main() {
  int i, iarray[8];

  iarray[0] = 1;
  iarray[3] = 7;
  i = 2;
  iarray[2*i+1] = iarray[3];
}
```

Notice a few things:

- Array elements start from zero and go up to **one less than** the numbers of elements in the array.
- We can use an array element anywhere we may use an ordinary variable.
- The contents of the square brackets [] may be any integer expression provided its value lies within the legal range.
- This example leaves the values of array elements 1,2,4,6 and 7 completely undefined.

When we specify an array of length, say, ten the machine reserves a contiguous block of memory large enough to store the values of ten variables - forty bytes in this case. `iarray[0]` then refers to the first four byte chunk, `iarray[1]` to the second four byte chunk, and so on:

```
               -------------------------------------------------
Byte number:   | 600 | 601 | 602 | 603 | 604 | 605 | 606 | 607 |
               -------------------------------------------------
Used for:      | <---- iarray[0] ----> | <---- iarray[1] ----> |
               -------------------------------------------------
```

This is why we can use an array element anywhere where we use a variable - to the computer they are both just ways of specifying a chunk of memory and it doesn't care how we did it.

Multidimensional arrays

Arrays can have as many dimensions as we like as in this snippet:

```
  int a[3][2];  /* Two dimensional array of integers */

  a[2][1] = -10;
```

If the compiler decide to store the array starting at byte number forty in its memory the elements of the array would then stored in memory as:

```
   -----------------------------------------------------------------
   | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 |
   -----------------------------------------------------------------
   | <--- a[0][0] ---> | <--- a[0][1] ---> | <--- a[1][0] ---> | etc.
   -----------------------------------------------------------------
```

Notice the order it stores them in: just as the number 10 is followed by 11 not 21 so a[1][0] is followed by a[1][1] not a[2][0].

NB: statically-declared multidimensional arrays like these are used much less than dynamically allocated ones which we discuss in a later lecture.

#define

This has a variety of uses, one simple one is to give a name to a constant so that it can be changed later. In this example we have an array and a `for()` loop to go over the whole aray:

Note: no semi-colon.

```
#define N 20

int main() {
  double values[N];
  int i;

  for (i = 0; i < N; ++i) {
    ...
```

```
  }
  return 0;
}
```

(We remind ourselves of the `for()` loop below.)

Characters and strings

In C single characters are contained in single quotes (`'a'`), strings of characters in double quotes(`"abc"`). The special value zero (`'\0'`) means "end of string". (`'\0'` is very different from `'0'`!)

Strings are stored as an array of characters:

```
char string[4] = "abc";
```

Notice that `string` is an array of length four, not three. Its elements have the following values:

string[0]	'a'
string[1]	'b'
string[2]	'c'
string[3]	'\0'

How characters are stored

Internally, characters are stored as integers. But it is a mistake to rely on this and write code such as:

```
char letter = 65;
```

Instead we should always use the actual character, inside single quotes:

```
  char letter = 'A';
```

for one simple reason: it is much clearer.

Arrays of strings

Since a character string is just a one-dimensional character array, it follows that an array of strings is a two-dimensional character array. The following allows us to store up to twenty strings each of length up to forty characers long.

```
#define STRINGLEN 40
#define MAXSTRINGS 20

int main {
  char mystrings[MAXSTRINGS][STRINGLEN + 1];

  strncpy(mystrings[0], "Hello world", STRINGLEN);
}
```

If statements

An `if` statement consists of a logical test followed by a block of statements that are only executed if the test is true (ie non-zero). The simplest if statement looks like this:

```
  if (i == j) {
    /* Statements here only get executed if i is equal to j */
  }
```

(The notation == for "logical equals" is slightly confusing:
`i = j` means "make i equal to the current value of j", whereas
`i == j` is a test for "is i equal to j?".)

If the statement in parentheses after the 'if' is true the statements inside the curly brackets (called *braces*) are executed, if not they are ignored.

It is possible to specify alternatives if the first statement is not true:

```
  if (i == j) {
    /* statements here */
  }
  else if ( i < j ) {
    /* More statements here */
  }
  else if (i == 0 && j > 1 ) {
    /* More statements here */
  }
  else {
    /* Only executed if all the others fail */
  }
```

Note:

- The braces are optional if the content of an if block consists of just a single statement.
- The `else if` and `else` statements are optional
- `if .. else if` statements specify **alternatives** - if more than one test is true only the **first** is evaluated - the rest are skipped.

A common mistake

```
  if (i = j) {
    /* statements here */
   }
```

This is perfectly legal, what does it do?

Relational operators

C provides the following relational operators:

<table>
<tr><td>!</td><td>not, e.g. ! (i == j) is equivalent to i != j)</td></tr>
<tr><td><   <=   >   >=</td><td>less than, less than or equal to, etc</td></tr>
<tr><td>==   !=</td><td>equal to, not equal to</td></tr>
<tr><td>&&</td><td>and</td></tr>
<tr><td>||</td><td>or</td></tr>
</table>

eg:

```
if ( b*b > 4*a*c && a != 0 )
  return (-b + sqrt(b*b - 4*a*c))/(2*a);
```

In the above list operators on the top lines have higher precedence (bind more tightly) than the lower lines. Items on the same line have equal precedence. For example:

```
 i == j && j < 0 ||  i != j && j > 0
```

is equivalent to:

```
 (i == j && j < 0) || (i != j && j > 0)
```

C doesn't bother to have a separate data type for logical expressions, it just treats them as integers. The logical expressions above return the value one if they are true and zero if they are false. Conversely, C treats any

integer expression with a value of zero as being false, any other value is treated as true. If the expression given is not an integer it is converted to one.

The switch statement

The switch statement chooses between one of several (integer) alternatives.

```
switch (i) {

case 'a':
  printf("a\n");
  break;

case 'b': case 'c':
  printf("b or c\n");
  break;

case 'd':
  printf("d\n");
  /* Fall through */

case 'e':
  printf("d or e\n");
  break;

default:
  printf("Something else\n");
}
```

Something of a mis-feature is the fact that by default control falls through to the next case (as in the case `'d'` above). Personally, if I want this to happen I comment it as above.

Loops

The while loop

A while loop looks like this:

```
while (expression) {
  /* Commands in here */
}
```

Here expression is any integer expression, eg `i+j` or `k != m`, just as in the `if` statement above. When the program runs, if the value of the expression is zero the entire contents of the braces {} are skipped, if the value of the expression is anything except zero they are executed. In the latter case the compiler then goes back, re-evaluates the test expression and tries again. Naturally, if the contents of the loop do nothing to change the value of the controlling expression the loop will continue for ever!

Steps for the while loop

1. Evaluate the test expression.
2. If the test was true, execute the statement block, otherwise quit.
3. Go back to 1.

The do .. while loop

The `do .. while` loop does the same as the `while` loop but in the opposite order in that the block of commands is executed before the test is made. Thus the contents are always run once even if the test is false.

```
do {
  printf("Please enter an integer from 1 to %d\n", j);
  scanf("%d", &i);
} while (i < 1 || i > j);
```

The `do .. while` loop is not used very much, the above is the most common case.

Steps for the do .. while loop

1. Execute the statement block.
2. Evaluate the test expression.
3. If the test was true, go back to 1, otherwise quit.

The for loop

The for loop consists of an initialisation, a test and an update (which is executed after the statement block) all in one statement. We will show this loop with a single statement in the loop, allowing us to omit the optional braces:

```
  for (i = 0; i < 10; ++i)
    iarray[i] = i*i;
```

This is a classic use of the `for` statement - doing something on each element of an array. Notice:

- Just like the `while` loop, the test is done before the statement block is executed.
- If the test is false to start off with then neither the statement block nor the update are executed.
- The order within the `for` statement is the order in which they occur:
  `for (initialisation; test; update)`

  Steps for the for loop

  1. Perform the initialisation.
  2. Evaluate the test expression.
  3. If the test was false quit.
  4. Otherwise, execute the statement block and
  5. perform the update operation and
  6. go back to 2.

Neither of these loops does anything that we couldn't do with a `while` loop, it's just that the situations they describe occur sufficiently frequently to make it worth having special statements for them.

Infinite loops

Either of these gives us a loop that never finishes:

```
while (1) {
  /* Infinite loop here */
}
```

```
for(;;) {
  /* Infinite loop here */
}
```

The first is more intuitive but the second is the more commonly used. It uses the fact, not mentioned before, that any of the three elements of the `for` statement may be omitted and that if the logical test is omitted it defaults to `TRUE`.

Functions

The main purpose of a function is to hide complexity from the person who is calling it, and to a lesser extend from the person who is writing it. Examples include:

- Tasks where it is easy to describe what we want it to do, but harder to describe how to do it.
- Things we will need to do several times.
- Things we are sufficiently complicated and self-contained that it's easier to do that task (and test it out!) without having to think about the "big piture".

A typical function in C looks something like:

```
float fun1(int n, float x) {
  int i;
  float z, myarray[2*n];
```

```
  /* Main body of function here */

  return(z);
}
```

and can be called from another function as part of an expression as in this example:

```
   a = x +  fun1(m, b);
   y = x +  fun1(2, sin(theta) );
```

This particular function returns a value (a float) but others do not. In C even the main routine is a function, called `main()`.

Variables inside functions disappear when the function returns

None of the variables defined within the function (in this case the variables `i` and `z` and the array `myarray`) existed before this function was called and when the function returns they will go away again and their values can never be recovered. If `fun1` is called again later they will **not** start off with the values they had when they last left `fun1`.

Variables defined **inside** functions are called *automatic* variables because they are automatically created and destroyed as needed.

A classic bug

```
char *badfun(some args) {
   char string[12];

   /* Write something to the string */
   return string;
}
```

Functions receive copies of their arguments

In the above example, when we called `fun1` the values of `m` and `b` in the calling routine are **copied** to new temporary variables `n` and `x` inside `fun1`. When `fun1` exits `n` and `x` will be lost along with `i` and `myarray`. This is more obvious in the second example where it's hard to even imagine how we could change the values of `2` and `sin(theta)`.

```
#include <stdio.h>

void addem(int i) {
  i = i + 2;
  printf("Inside addem i is now: %d\n", i);
}

int main() {
  int i;

  i = 317;
  printf("Before addem i is: %d\n", i);
  addem(i);
  printf("After addem i is: %d\n", i);
}
```

Passing arrays to functions

The value of the name of an array is the value of the location in memory where the array is stored.

So, in our previous example of an array called `iarray` stored at memory location 600, if we were to pass `iarray` to a functioon we would be passing the number "600". Thus:

```
void myfun(int n, jj[n]) {
  int k;

  for(k = 0; k < n; ++k)
    jj[k] = k *k;
```

```
}
...

// inside main()
   myfun(N, iarray);
```

Since `jj` inside `myfun()` has the value 600, it follows that `jj[0]` refers to the integer stored at location 600, `jj[1]` to that at 604, etc, i.e. the original array inside `main`.

Automatic variables are very convenient

- The calling function is easier to understand because there is no danger that `fun1` can change either `m` or `b`.
- The machine takes care of the memory bookkeeping for us.

  Computers only have a limited amount of memory and if every time fun1 was called it created a brand new set of variables and didn't destroy them when it returned then the amount of memory being used by the program would increase every time fun1 was called until the computer ran out of memory. By deleting them when the program leaves the function it can reuse the same memory time and time again.
- fun1 is able to call itself and each invocation has its own values of all its variables, etc. This is known as *recursion* (as in this simple example on the Web).
- The process of allocating and deallocating space for these variables is very quick.

Automatic variables have their limitations

- There is no way fun1 can change the values any of the variables in its calling routine. Although we don't *normally* want to do that there is always the odd occasion when we do want to. We have already met this problem using scanf when the answer was to put '&' in front of the variable name and later we will discuss what this means.
- Sometimes a routine does need to create a permanent new variable or array which can then be used by the rest of the program and automatic variables cannot do that.

We note in passing that many of limitations, and more importantly all of these advantages, do not apply to variables defined **outside** of functions. These are called external variables and we will discuss them a little later in the course.

Prototypes check function arguments

It's surprisingly easy to get the arguments to a function wrong when calling it. For example we might miss one out, or get them in the wrong order. A little later we may add a new feature to the function which causes it to require another argument but we might forget to add this argument to one of the places where it is called.

Fortunately, we can get the compiler to check that we have the right number of arguments and that they are in the right order:

- If the function and all the places it is called are in the same file and the function appears before all the places it is called then the compiler will do this automatically.
- If the function is in a different file from where it is called, create a file called, say, decls.h, containing the line:

  ```
  float fun1(int n, float x);
  ```

  This is called a *prototype* and it tells the compiler what type of arguments fun1 requires and what type of value it returns. If we prefer we can miss out the 'n' and the 'x' and just write `float fun1(int, float);` . Now in the file where fun1 is defined and in all the files where fun1 is called add the line:

  ```
  #include "decls.h"
  ```

  at the top. The compiler will behave as if the contents of decls.h had actually been written in the file at that point and will use the prototype to check the arguments. Naturally, we should do this for each of the functions we define.
- If the function and all the places it is called from are both in the same file but you prefer to have the calling routine first you can either use the previous method or just add the prototype before the first function definition in the file.

If you fail to use one of these methods to ensure that the compiler checks your arguments for you you

will lose marks.

Of course, this checking has one obvious limitation: if a function requires two arguments of the same type the compiler has no way of knowing if we have put them in the right order or not.

The standard library

C comes with a large collection of useful functions, each of which needs an **`include`** file, for example `#include<math.h>` allows use of `sin()`, etc.

Input and output

The **`printf()`** function prints to the screen:

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

  int main() {
    printf("The square root of %d is %g\n", 2, sqrt(2));
    return 0;
}
```

Note the use of `%g` for printing a double, %e and %f are also available but less useful. %i and %d ("decimal") print integers and are the same.

Input

The **`scanf()`** function reads values in:

```
  #include <stdio.h>

  int main() {
    int values;

    printf("Please enter the number of values\n");
    scanf("%d", &values);

    /* Do something interesting ... */

    return 0;
}
```

Note the ampersand before the variable name.

- **Gotcha**: the format for reading a double is **`%lf`** ("long float") as simple `%f` reads a float.

Using files

Using the **`FILE *`** allows reading from or writing to a file:

Notice that we checked that the call to `fopen()` succeeded before calling `printf()`.

Validating input

When input values must satisfy certain conditions, it's a good idea to check that they do! Two ways to do this are the `do...while()` loop or an infinite loop with a **`break`** statement if the input is correct. We illustrate the latter here:

Without the `break` statement the loop would never exit as of course one is always equal to one.

Multiple tests

One nice feature of the above idea is that we can test for multiple error conditions one at a time, with a separate message for each. For examplem, we can add a test to make sure that square has not already been played:

(Notice how we check that `x` and `y` are in the right range **before** checking that the grid is empty.)

Other functions

C's standard library provides several catagories of utlity functions. Here we just provide some links to the PHY2004 lecture notes.

- Mathematical functions **`<math.h>`**
  `sin()`, `cos()` ,`sqrt()`, etc. Remember to use **`<complex.h>`** for the complex version.
- Single character functions: **`<ctype.h>`**
  Character tests such as `isalpha()`, `isupper()`, `islower()`, `isdigit()`, `isalnum()`, `ispunct()`, etc.
  Character conversions: `toupper()`, `tolower()`
- String functions: **`<string.h>`**
  `strcmp(str1, str2)`, `strncpy(str1, str2,N)`, `strchr(str, c)`, etc.
- **`<stdlib.h>`**
  Currently we have only used this for `exit()`.

Things to remember

Mega-principle

Whenever we are faced with a choice, our first question should always be:

"**which choice will be the clearest and give me the least chance of making a mistake?**".

- In general, the structure and flow of our code and the names of our variables should mirror the most natural description of what we are trying to do.
- **Correct indentation** is vital for understanding our code.
- **Comments** are extremely useful just before the start of a function, but only useful inside a function when there is something that cannot be made clear by the code itself.
- Avoid possibilities for ambiguity. This is why comments before a function are so useful.

Don't get fooled by "big picture" messages

Consider the following:

```
if ( x < 0 )
    printf("X is negative, correcting\n");
    x = -x;

value = sqrt(x);
```

It's clear what we wanted to do, but that isn't what will happen.