C Basics

Wednesday, 11 April 2018 8:42 PM

All information for this course will come from Andrew Taylor's and Andrew Bennett's slides/notes

Variables

Variables are used to store a value. The value a variable holds may change at any time. At any point in time a variable stores **one** value, (except for quantum computers).

C variables have a type. For this course the main variables we use are:

- `int` - for integer values
- `double` - for decimal numbers
- `char` - for characters

Integer Representation

Typically 4 bytes are used to store an **int** variable.
4 bytes -> 32 bits -> 2^{32} possible values (bit patterns)
This means only 2^{32} integers can be represented.
These integers are: -2^{31} to $2^{31} - 1$ (-2,147,483,648 to +2,147,483,647)
These limits are asymmetric because zero needs a pattern (the pattern being all zeros).

Integer Overflow/Underflow

Storing a variable in an **int** outside the range that it can be represented is *illegal*. This can result in unexpected behaviour from most C implementations, or it may cause programs to halt, or not terminate. This can increase security holes.

Bits used for **int** can be different on other platforms. For example, C on a tiny embedded CPU in a washing machine may use 16 bits. For now we assume **int** uses 32 bits.

Real Representation

Commonly 8 bytes are used to store a **double** variable.
8 bytes -> 64 bits -> 2^{64} possible value (bit patterns)
64 bits give huge number of patterns but infinite number of reals.

More on Variables

How to use variables:

1. Declare
2. Initialise
3. Use

```
int number = 0;
printf("%d\n", number);
```

Note: steps 1 and 2 can usually be done in the same line.

If you don't initialise a variable, it will take the last value that was stored in that variable, and this can lead to bad stuff :(

Variable names:
Variable names can be made up of letters, digits and underscores.
Here are some unwritten rules about variable names:

- Use a lowercase letter to start your variable name
- Beware! variable names are case sensitive
- Beware! certain words can't be used as variable names; e.g. **if**, **while**, **return**, **int**, **double**. This is because these keywords have special meanings in C programs

Printing and reading variables:
Use **%d** to read and print an **int** value
Use **%lf** to read and print a **double** value
Note: if you want specific decimal places for **double** write it like this: **%.nlf**, where n is how many decimal places
The format string is of the form

% [*flags*] [*field_width*] [*.precision*] [*length_modifier*] *conversion_character*

where components in brackets **[]** are optional. The minimum is therefore a **%** and a conversion character (e.g. **%i**).

Flags

Flag	Meaning
-	The output is left justified in its field, not right justified (the default).
+	Signed numbers will always be printed with a leading sign (**+** or **-**).
space	Positive numbers are preceded by a space (negative numbers by a **-** sign).
0	For numeric conversions, pad with leading zeros to the field width.
#	An alternative output form. For **o**, the first digit will be **'0'**. For **x** or **X**, **"0x"** or **"0X"** will be prefixed to a non-zero result. For **e**, **E**, **f**, **F**, **g** and **G**, the output will always have a decimal point; for **g** and **G**, trailing zeros will not be removed.

Field width

Converted argument will be printed in a field at least this wide, and wider if necessary.
If the converted argument has fewer characters than the field width, it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally **' '** (space), but is **'0'** if the zero padding flag (**0**) is present.
If the field width is specified as *****, the value is computed from the next argument, which must be an **int**.

Precision

A dot **'.'** separates the field width from the precision.
If the precision is specified as *****, the value is computed from the next argument, which must be an **int**.

Conversion	Meaning
s	The maximum number of characters to be printed from the string.
e, **E**, **f**	The number of digits to be printed after the decimal point.
g, **G**	The number of significant digits.
d, **i**, **o**, **u**, **x**,**X**	The minimum number of digits to be printed. Leading zeros will be added to make up the field width.

Length modifier

Character	Meaning
h	The value is to be displayed as a **short** or **unsigned short**.
l	For **d**, **i**, **o**, **u**, **x** or **X** conversions: the argument is a **long**, not an **int**.
L	For **e**, **f**, **g** or **G** conversions: the argument is a **long double**.

Conversion character

Character	Meaning
d, **i**	Display an **int** in signed decimal notation.
o	Display an **int** in unsigned octal notation (without a leading **0**).
u	Display an **int** in unsigned decimal notation.
x, **X**	Display an **int** in unsigned hexadecimal notation (without a leading **0x** or **0X**). **x** gives lower case output, **X** upper case.
c	Display a single **char** (after conversion to **unsigned int**).
e, **E**	Display a **double** or **float** (after conversion to **double**) in scientific notation. **e** gives lower case output, **E** upper case.
f	Display a **double** or **float** (after conversion to **double**) in decimal notation.
g, **G**	**g** is either **e** or **f**, chosen automatically depending on the size of the value and the precision specified. **G** is similar, but is either **E** or **f**.
n	Nothing is displayed. The corresponding argument must be a pointer to an **int** variable. The number of characters converted so far is assigned to this variable.
s	Display a string. The argument is a pointer to **char**. Characters are displayed until a **'\0'** is encountered, or until the number of characters indicated by the precision have been displayed. (The terminating **'\0'** is not output.)

p	Display a pointer (to any type). The representation is implementation dependent.
%	Display the **%** character.

Above information from <http://personal.ee.surrey.ac.uk/Personal/R.Bowden/C/printf.html>

Giving Constants Names

It can be useful to give constants names. In general any *magic* number that holds meaning should be given constant names. This makes your program *readable*. It can also make your program easier to update especially if the constant appears in many places.

One method of doing this is the **#define** statement. **#define** statements go at the **top** of your program after the **#include** statements and **#define** names should always be in capital letters and underscores.

```
#define NAME_OF_CONSTANT 100
```

Mathematics in C

C supports the usual maths operations: + - * /. BODMAS follows.

Beware! Division in C is not what you'd expect
When dividing with **double**, C division is what you'd expect: 2.6/2 = 1.3
When dividing with **int**, C gets a bit weird: 2.6/2 = 1. The fraction part is discarded (not rounded).

C also has the **%** (modulo) operator, but it is for integers only.

Mathematical functions are not part of the standard library because tiny CPUs may not support them.
The library **math.h** contains mathematical functions such as `sqrt(), sin(), cos(), tan()` (These take **double** as arguments and return **double**).

If Statements

Wednesday, 11 April 2018 8:43 PM

Many problems require executing statements only in some circumstances. This is sometimes called **control flow, branching** or **conditional execution**. We do this in C using **if** statements.

A general if statement appears like this:

```
if (expression1) {
     statement1;
     …
} else if (expression2) {
     statement2;
     …
} else {
     statement3;
}
```

statement1, … are executed if **expression1** is non-zero.
statement1, … are not executed in **expression1** is zero.
Instead it will check if **expression2** is non-zero, and the same thing happens as **expression1.**
If none of the expressions are non-zero, **else** will execute **statement3**.

Also, there is no "boolean" type in C.
0 is regarded as FALSE
Anything non-zero is regarded at TRUE

Relational Operators

C has the usual operators to compare numbers:

Operator	Meaning
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
!=	not equal to
==	Equal to

Be careful when comparing **double**s for equality == or !=. Recall that **double**s are approximations.

Many languages have a separate type for true & false. C just uses 0 for false and other numbers for true

Relational operators return:
The **int 0** for false
The **int 1** for true

Logical Operators

C also has logical operators:

Operator	Meaning
&&	**and** operator - true if both operands are true
||	**or** operator - true if either operand is true
!	**not** operator - true if and only if its operand is false

Functions

Wednesday, 11 April 2018 8:42 PM

Functions allow you to:

- separate out "encapsulate" a piece of code serving a single purpose
- test and verify a piece of code
- reuse the code
- shorten code resulting in easier modification and debugging

Some functions we already use are:
`printf()` and `scanf()`

Structure of a Function

1. Return type
2. Function name
3. Parameters (inside brackets, comma separated)
4. Variables (which are only accessible within the function)
5. Return statement

Example:

```
int add_numbers(int num1, int num2) { // 1, 2, 3
     int sum; // 4
     sum = num1 + num2;
     return sum; // 5
}
```

The `return` Statement

When a **return** statement is executed, the function terminates. The **return** expression will be evaluated, and if necessary, converted to the type expected by the calling function.

All local variables and parameters will be thrown away when the function terminates. The calling function is free to use the return value, or to ignore it.

Functions can be declared as **void**, which means that nothing is returned. A **return** without a value statement can still be used to terminate such a function.

Calling a function

When calling a function, you type the function name and the variable you will pass in to the function.
E.g. x = `cube(number);`

- cube is the function name
- number is the variable we are passing into the function.

Function Properties

- Functions have a type. This is the type of value they return.
- Type **void** is for functions that return no value. **void** is also used to indicate that a function has no parameters.
- Functions cannot return arrays
- Functions that have their own variables created when the function is called and these variables are destroyed when the function returns.
- A function's variables are not accessible outside the function
- **return** statements stop the execution of a function
- **return** statements specify the value to return unless the function is of type void
- A run-time error occurs if the end of a non-**void** function is reached without a **return**

Function with No Return Value

Some functions do not compute a value.
They are useful for "side-effects" such as output.

```
void print_sign(int b) {
        if (b < 0) {
                printf("negative");
        } else if (b == 0) {
                printf("zero");
        } else {
                printf("positive");
        }
}
```

Function Parameters

Functions take 0 or more parameters.
Parameters are variables created each time the function is called and it is destroyed when the function returns. C functions are *call-by-value* (but beware arrays). The parameters are initialised with the value supplied by the caller. The value is essentially copied to the parameter. A parameter variable changed in the function has no effect outside of the function.

Function Prototypes

Function prototypes allow function to be called before it is defined. It species key information about the function:

- function return type
- function name
- number and type of function parameters

It allows a top-down order of functions in the file, which makes it more readable. It also allows us to have function definition in a separate file. This is important since it is crucial to share code and important for larger programs

Library Function

Over 700 functions are defined in the C standard library.

The C compiler needs to see a prototype for these functions before you use them. You do this indirectly with **#include** line. For example **stdio.h** contains prototypes for **printf** and **scanf.**

While Statements

Wednesday, 11 April 2018 8:42 PM

if statements only allow us to execute or not execute code. This means that they execute code either once or never. Meanwhile, **while** statements allow us to execute code once or more times.

Like **if**, **while** statements have a controlling expression but **while** statements execute their body of statements until the controlling expression is false.

The general construction of a while statement looks like this:

```
while (expression) {
     stmt1;
     stmt2;
     …
     stmtn;
}
```

Loop Counter

Often we use a **loop counter** variable to count loop repetitions. This allows us to have a **while** loop execute **n** times.

An example of using the loop counter:

```
// read an integer n
// print n asterisks
int loop_counter, n;

printf("How many asterisks? ");
scanf("%d", &n);

loop_counter = 0;
while (loop_counter < n) {
     printf("*");
     loop_counter++;
     // this is the same as loop_counter = loop_counter + 1;
}
printf("\n");
```

Termination

We can control the termination (stopping) of **while** loops in many ways. However, it is very easy to write a **while** loop which does not terminate. Often a **sentinel** variable is used to stop a **while** loop when a condition occurs in the body of the loop.

Here is a general outline for using the sentinel variable pattern:

```
stop_loop = 0;
while (stop_loop != 1) {
     //
     // statements the loop needs to perform
     //
     if (.......) {
          stop_loop = 1;
     }
     //
     // perhaps more statements
     //
}
```

Nested While Loops

We often need to nest **while** loops. When we do this we need separate counter variables for each nested loop.

```
// print a square of 10x10 asterisks
int i, j;
i = 0;
while (i < 10) {

     j = 0;
     while (j < 10) {
          printf("* ");
          j = j + 1;
     }

     printf("\n");
     i = i + 1;
}
```

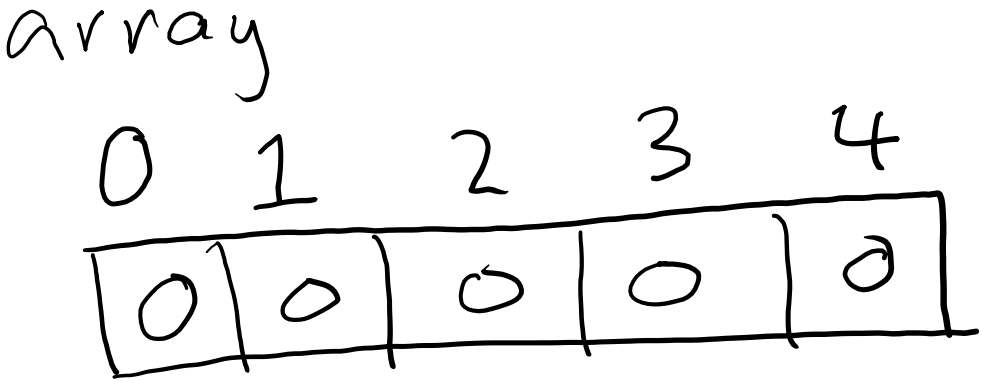
Arrays

Wednesday, 11 April 2018 8:42 PM

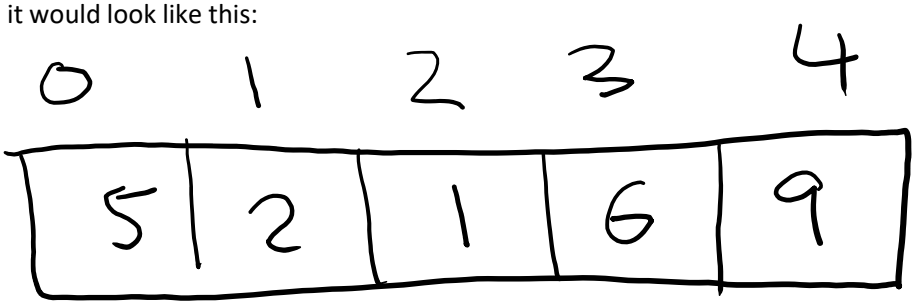
A C **array** is a collection of variables called **array elements**. All array elements must be of the same type. Array elements do not have names, instead they are accessed by a number called the **array index**. A valid array index for an array with *n* elements are:
0, 1, 2, … , *n* - 1
Arrays must be initialised, or else weird stuff will happen, which you don't want.

You also can assign scanf or printf whole arrays, instead you can assign scanf/printf array elements.

```
// Declare and initialise the array
int array[5] = {0};
```
Diagrammatically the array would look like this:




```
// You can also initialise the array like this
int array[5] = {5, 2, 1, 6, 9};
```
In a diagram it would look like this:



Reading and Printing Arrays

When reading arrays, you must read each element individually. This is easiest to do with a **while** loop.

```
#define ARRAY_SIZE 20
i = 0;
while (i < SIZE) {
    scanf("%d", &array[i]);
    i = i + 1;
}
```

Likewise, if you are printing arrays, you must print each element individually

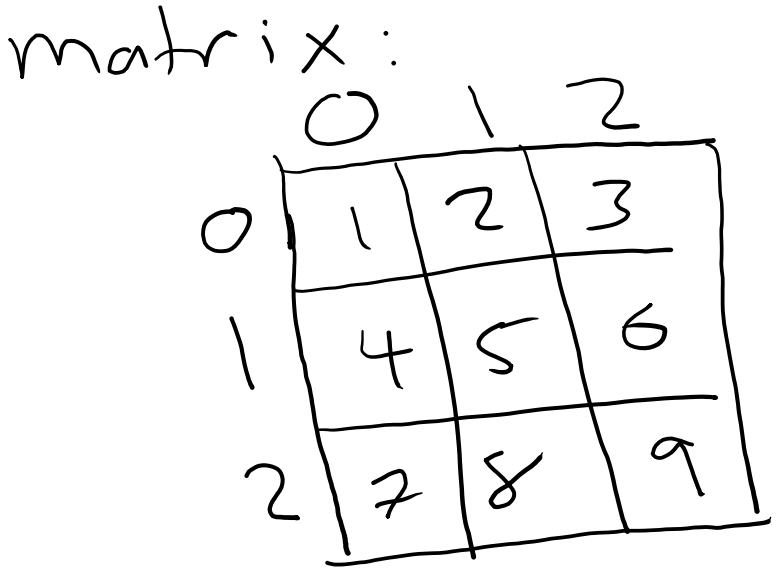
```
i = 0;
while (i < ARRAY_SIZE) {
    printf("%d\n", array[i]);
    i = i + 1;
}
```

Arrays of Arrays/ Matrixes

C supports arrays of arrays. This is useful for multi-dimensional data.

A two-dimensional array (a matrix) would be useful for storing both rows and columns of data.
Declaring and initialising an array would look like this:

```
int matrix[3][3] = { {1, 2, 3},
                     {4, 5, 6},
                     {7, 8, 9} };
```



Reading and Printing Two-dimensional Arrays

When reading two-dimensional arrays, we need to use a nested **while** loop, which uses two counters; One for the row and the other for the column.

```
#define SIZE 42
...
int matrix[SIZE][SIZE];
int i, j;

i = 0
while (i < SIZE) {
    j = 0;
    while (j < SIZE) {
        scanf("%d", &matrix[i][j]);
        j = j + 1;
    }
    i = i + 1;
}
```

The same applies for printing arrays.

```
...
while (i < SIZE) {
    j = 0;
    while (j < SIZE) {
        print("%d", &matrix[i][j]);
        j = j + 1;
    }
    printf("\n");
    i = i + 1;
}
```

Strings

Wednesday, 11 April 2018 8:42 PM

The char type

The C type **char** stores small integers. It is almost always 8 bits. **char** is guaranteed to be able to represent integers 0..+127 and it is most likely to store ASCII character codes.

To print **char**s with printf we use **%c.**

Do not use **char** for individual variables. Use them only for arrays and characters. Even if a numeric variable is only used for values 0..9, use the type **int** for the variable.

ASCII Encoding

ASCII stands for **A**merican **S**tandard **C**ode for **I**nformation **E**xchange. It specifies the mapping of 128 characters to the integers 0…127. The characters encoded include:

- Upper and lower case English letters: A-Z and a-z
- Digits: 0-9
- Common punctuation symbols
- Special non-printing characters: e.g. newline and space

But don't worry you don't have to memorise ASCII. Just writing single quotes give you the ASCII code for a character

```
printf("%d", 'a'); // prints 97
printf("%d", 'A'); // prints 65
printf("%d", '0'); // prints 48
printf("%d", ' ' + '\n'); // prints 42 (32 + 10)
```

So don't put ASCII in your programs - just use single quotes.

Reading a character - getchar

C provides library functions for reading and writing characters.
getchar reads a byte from standard input and returns and int. **getchar** returns a special value, **EOF** (usually **-1**) if it cannot read a byte. EOF stands for **E**nd **O**f **F**ile. Otherwise **getchar** returns and integer (0..255) inclusive. If standard input is a terminal or a text file, this is likely to be an ASCII code. Beware, input is often buffered before an entire line can be read.

End of Input

Input functions such as **scanf** or **getchar** can fail because no input is available. e.g. if input is coming from a file and the end of the file is reached. On UNIX-like systems (Linux/OSX) typing **ctrl** + **D** signals to the operating systems no more input from the terminal. Windows has no equivalent to this, although some windows programs interpret **ctrl** + **Z** similarly.

getchar returns a special value to indicate there is no available. This non-ASCII value is #defined as **EOF** in stdio.h. On most systems EOF == -1. There is no end-of-file character on modern operating systems.

The programming pattern for reading characters to the end of input would look like this:
Programming pattern for reading characters to the end of input:

```
int ch;

ch = getchar();
while (ch != EOF) {
    printf("'%c' read, ASCII code is %d\n", ch, ch);
    ch = getchar();
}
```

Strings

A string in computer science is a sequence of characters. In C, strings are an arrays of **char** containing ASCII codes. These arrays of char have an extra element containing a 0. The extra 0 can also be written '**\0**' and may be called a NULL character or NULL-terminator. This is convenient because programs don't have to track the length of the string.

Because working with strings is so common, C provides some convenient syntax.
Instead of writing:

```
char hello[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

You can write

```
char hello[] = "hello";
```

Note: hello will have 6 elements; 5 for the individual characters, 1 for the NULL terminator '\0'.

The C library includes some useful functions which operate on characters. Here are some below:

```
#include <ctype.h>

int toupper(int c); // convert c to upper case
int tolower(int c); // convert c to lower case
int isalpha(int c); // test if c is a letter
int isdigit(int c); // test if c is a digit
int islower(int c); // test if c is lower case letter
int isupper(int c); // test if c is upper case letter
```

Reading a string/line - `fgets`

`fgets(array, array size, stream)` reads a line of text

1. **array** - is a **char** type array. This is where the line will be stored.
2. **array size** - is the size of the array. This size is how big the line can be.
3. **stream** - is where the line will be read from e.g. **stdin.**

fgets cannot not store more characters than the array size, because it will not be big enough .
fgets always stores a terminating character ,'\0', in the array.
fgets stores a newline character, `\n', in the array, if it reads an entire line. We often need to overwrite this newline character:

```
int i = strlen(lin);
if (i > 0 && line[i - 1] == "\n) {
    line[i - 1] = '\0';
}
```

NEVER use the similar C function **gets,** which can overflow the array and cause major source of security exploits.

The programming pattern for using **fgets** looks like this:

```
#define MAX_LINE_LENGTH 1024
...
char line[MAX_LINE_LENGTH];
printf("Enter a line: ");
// fgets returns NULL if it can't read any
characters
if (fgets(line, MAX_LINE_LENGTH, stdin) != NULL {
    fputs(line, stdout);
    // or
    printf("%s" ,line); // same as fputs
}
```

Command-line Arguments

Command-line arguments are 0 more strings specified when the program is run.
If you run this command in a terminal:

```
$ dcc -o count  count.c
```

dcc will be given 3 command-line arguments: "count.c" "-o" "count"

Our main function will need different parameters if we want to access command-line arguments

```
int main(int argc, char *argv[]) { … }
```

argc stores the number of command-line arguments + 1.
If **argc == 1** , there are no command-line arguments.
argv stores the program name and the command-line arguments.
argv[0] always contains the program name
argv[1] argv[2] … will contain the command-line arguments if they are supplied

If we want to print out the command-line arguments we would do something like this:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i = 1;
    printf("My name is %s\n", argv[i]);
    while (i < argc) {
        printf("Argument %d is: %s\n", i,
        argv[i]);
        i = i + 1;
    }
}
```

Week 6 Tutorial

char is a data type that stores single characters
%c is used to read and print characters when using printf and scanf

`string.h` functions

Here are some string functions that may be useful:

```
#include <string.h>
// string length (not including '\0')
int strlen(char *s);

// string copy
char *strcpy(char *dest, char *src);
char *strncpy(char *dest, char *src, int n);

// string concatenation/append
char *strcat(char *dest, char *src);
char *strncat(char *dest, char *src, int n);

// string compare
int strcmp(char *s1, char *s2);
int strncmp(char *s1, char *s2, int n);
int strcasecmp(char *s1, char *s2);
int strncasecmp(char *s1, char *s2, int n);

// character search
char *strchr(char *s, int c);
char *strrchr(char *s, int c);
```

Converting Command-line Arguments

stdlib.h defines some useful functions for converting strings.

atoi (for ASCII to integer) converts strings to int
atof (for ASCII to float) converts strings to double

Here is a demonstration of how to use **atoi:**

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    int i, sum = 0;
    i = 1;
    while (i < argc) {
        sum = sum + atoi(argv[i]);
        i = i + 1;
    }
    printf("sum of command-line arguments=%d\n", sum);
}
```

When you initialise an array of characters use double quotes ""
e.g. `char string[8] = "";`
But when assigning specific ASCII values use single quotes
e.g.
```
char character = 'A';
   char string[9] = { 'C', 'O', 'M', 'P', '1', '5', '1', '1', '\0'};
   printf("%s\n", string);
```
Remember: a string is an array of characters with a NULL terminator at the end.

Instead of individually assigning each ASCII character into the array you can just do this:
```
char string[9] = "COMP1511";
printf("%s\n", string);
```

Command Line Arguments
If we use command line arguments in our program the parameters in our main function should change to
```
int main(int argc, char *argv[]) {
}
```
argc is an integer telling how many commands there are.
argv is an array of strings storing what the commands are.

The program name itself is an argument.
```
./caesar 10
argc = 2
argv[0] = "./caesar"
argv[1] = "10"
```

Usage: ./caesar <number>

atoi converts ASCII to integer
```
int num = atoi(argv[1]);

if (argc != 2) {
     perror("Usage: %s <number\n", argv[0]);
     return 1;
}
```

putchar() and getchar()
getchar() receives characters
putchar() prints out characters

```
int number = getchar();
while (number != EOF) {
     putchar(number);
     number = getchar();
}
```

When we type in characters in terminal it doesn't immediately print out the characters. This is because the characters are held in a buffer waiting to be sent to the program once you press enter.

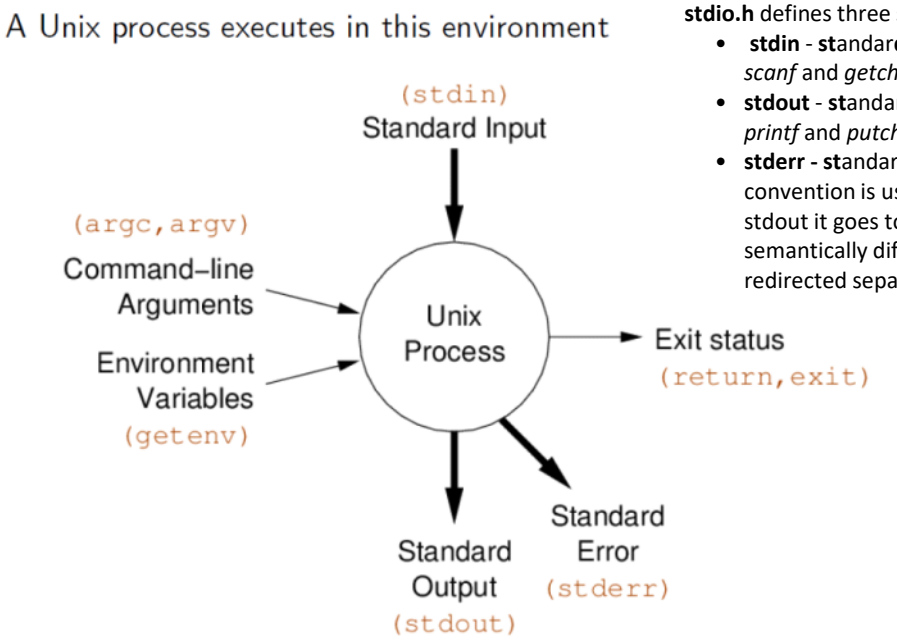
Cracking the caesar cipher: BRUTE FORCE!!!! There are only 25 shifts
Check against English words

I/O, Reading and Writing Files

Wednesday, 11 April 2018 2:04 PM

Introducing I/O (Input/Output)

The Unix process environment can be represented like this:



Can access with ***fprintf***. This is just like printf but works with files

Standard Streams

A stream is a sequence of bytes.
stdio.h defines three streams:

- **stdin** - **st**andar**d in**put stream; functions like *scanf* and *getchar* read from stdin
- **stdout** - **st**andar**d out**put stream; functions like *printf* and *putchar* write to stdout
- **stderr - st**andar**d err**or stream; this stream by convention is used for error messages; like stdout it goes to the terminal but it is semantically different to stdout; it can be redirected separately to **stdout**

I/O Direction

If a program runs from terminal:
stdin, **stdout**, **stderr** are connected to the terminal
Unix shell allows you to re-direct **stdin**, **stdout** and **stderr**

Run **a.out** with **stdin** coming from file **data.txt**
This will import the content from data.txt to a.out

```
./a.out < data.txt
```

Run **a.out** with **stdout** going to (overwriting) file **output.txt**
This is export the content from a.out to output.txt

```
./a.out > output.txt
```

Run **a.out** with **stdout** appended to file **output.txt**
This will add content from a.out to output.txt

```
./a.out >> output.txt
```

Run **a.out** with **stderr** going to file **error.txt**

```
./a.out 2> error.txt
```

Unix Pipes

Unix shell allows you to connect **stdout** of one program to **stdin** of another program.

Run **a.out** with **stdout** going to stdin of **wc**

```
./a.out | wc -l
```

wc counts chars, words and lines in its *stdin*
Some other useful Unix programs (a.k.a filters) designed to be run like **wc** include **grep**, **cut**, **sort**, **uniq**.
This is covered in more detail in COMP2041.

Using **stderr** for error messages

fprintf allows you to specify which stream to print out to.
For example:

```
fprintf(stderr, "error: cannot open %s\n, f);
```

printf is actually just calling *fprintf* and specifying **stdout**

```
fprintf(stdout, …);
```

It is best for error messages to be written to **stderr**, so users see them even if **stdout** is directed to a file

Accessing Files in C

```
FILE *f = fopen(char *filename, char *mode);
```

This creates a stream to read from a file, or a stream to write to a file.
It returns **NULL** if it fails to open anything

```
int fclose(FILE *f);
```

This finishes operations on a file.
fclose is automatically called on a program exit, although some output may be cached until **fclose** is called.

```
int fgetc(FILE *f);
```

This reads a single character from a stream. It returns EOF if no character is available.

Opening a File

```
FILE *output_file = fopen("filename.txt", "w");
```

- `FILE *` is the type of the variable. It is a pointer to the file
- `fopen` - opens a file
- `parameter 1` - the name of the file to be opened
- `parameter 2` - the mode in which to open the file
- `return value` - a pointer to the file which has been opened. This pointer is then used to reference the opened file for operations such as reading, writing, and closing the file.
  The return value will be **NULL** if the file cannot be opened or if you try to open a file you don't have permission to access

```
int fgetc(FILE *f);
```

This reads a single character from a stream. It returns EOF if no character is available.

```
int fputc(int, FILE *f);
```

This writes a single character to a stream.

```
int fscanf(FILE *f, …);
```

This performs scanf from a stream and returns the number of values read.

```
int fprintf(FILE *f, …);
```

This prints to a specified stream.

Example:

```
// opens a file called output.txt in "writing" mode
FILE* output_file = fopen("output.txt", "w");

// prints "Hello!" to the file
fprintf(output_file, "Hello!\n");

// print to stdout (These are the same thing);
printf("Hello!");
fprintf(stdout, "Hello!");
```

Writing to a file

```
fputs("the text I want to put in the file\n", fp);
```

OR

```
fprintf(fp, "the text I want to put in the file\n");
```

OR

```
int c = "!";
fputc(c, fp);
```

The return value will be **NULL** if the file cannot be opened or if you try to open a file you don't have permission to access

fopen *mode* parameters

Mode	Description
"r"	opens an existing file for reading purposes
"w"	opens a text file for writing if it doesn't exist a new file is created start writing from the top of the file (i.e. it will overwrite the original file)
"a"	opens a text file for writing in appending mode if doesn't exist a new file is created starts writing at the end of existing file content
"r+" "w+" "a+"	opens a file for both reading and writing w+ truncates the file to zero if it exists a+ starts reading from the start of the file and writing at the end of the existing file contents

Reading from file

```
char line[MAX_LINE_LENGTH];
if (fgets(line, MAX_LINE_LENGTH, fp) != NULL) {
    printf("read line\n");
} else {
    printf("Could not read a line\n");
}
```

OR

```
int c;
c = fgetc(fp);
if (c != EOF) {
    printf("read a '%d'\n", c);
} else {
    printf("Could not read a character\n");
}
```

Memory and Pointers

Wednesday, 11 April 2018 3:14 PM

Decimal Representation

We can interpret the decimal number 4705 as:

$4 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$

The *base* or *radix* is 10, and the digits are 0 - 9

The place values:

...	1000	100	10	1
...	10^3	10^2	10^1	10^0

We can write the number as 4705_{10}. The subscript is used to denote the base

Binary Representation

In a similar way, we can interpret the binary number 1011 as:

$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

The *base* or *radix* is 2, and the digits are 0 and 1

The place values:

...	8	4	2	0
...	2^3	2^2	2^1	2^0

We can write the number as 1011_{10} (= 11_{10})

Hexadecimal Representation

We can interpret the hexadecimal number 3AF1 as:

$3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 10^0$

The *base* or *radix* is 16, and the digits are:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

The place values:

...	4096	256	16	1
...	16^3	16^2	16^1	16^0

We can write the number as $3AF1_{16}$ (= 15089_{10})

Binary to Hexadecimal

Convert 1011111000101001_2 to Hex:

1011	1110	0010	1001_2
B	E	2	9_{16}

Convert 10111101011100_2 to Hex:

0010	1111	0101	1100_2
2	F	5	C_{16}

Note: The two zeros in bold have been added

Hexadecimal to Binary

Convert each hex digit into the equivalent of 4-bit binary representation.

E.g. Convert $AD5_{16}$ to Hex:

A	D	5
1010	1101	0101_{16}

To print hexadecimal values in C use the place value holder %x

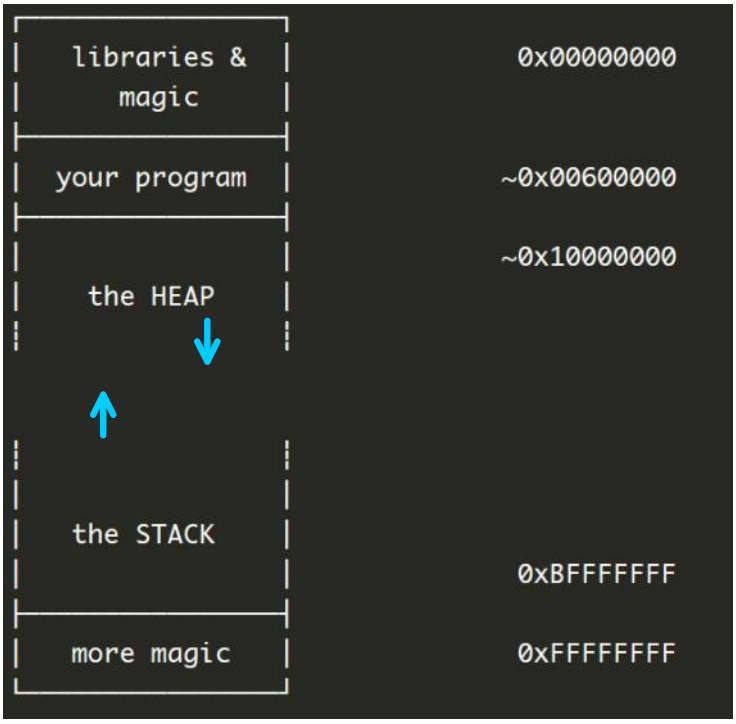
Memory Organisation

Memory is effectively a GIANT array of bytes. When a program is executed, program variables are stored in memory. Everything is stored in memory *somewhere*. Since everything is stored in memory, everything has an address.

Memory addresses are effectively an index to this array of bytes. There indexes can be very large - up to $2^{32} - 1$ on a 32-bit platform, and up to $2^{64} - 1$ on a 64-bit platform. Memory addresses are usually printed in hexadecimal (base 16).

In order to fully understand how pointers are used to reference data in memory, here's a few basics on memory organisation:

- Computer memory is a large array of *bytes*
- A variable will be stored in one or more bytes
- On CSE machines **char** occupies 1 byte, **int** occupies 4 bytes, **double** occupies 8 bytes
- The **&** operator returns address-of a variable. We sometimes call this a ***reference***. i.e &number is a ***reference*** to the variable "number"
- Almost all C implementations implement pointer values using a variable's address in memory. Hence for almost all C implementations, **&** operator returns a memory address.
- It is convenient to print memory addresses in Hexadecimal notation.
- Variables are stored on **the stack** (see diagram)
- Dynamic memory is stored on **the heap** (see diagram)



Variables in Memory

```
int k;
int m;
```

Arrays in Memory

The elements of an array will be stored in ***consecutive*** memory locations.

Variables in Memory

```
int k;
int m;
printf( "address of k is %p\n", &k );
// prints address of k is 0xbffffb80
printf( "address of m is %p\n", &m );
// prints address of k is 0xbffffb84
```

k occupies the 4 bytes from 0xbffffb80 to 0xbffffb83
m occupies the four bytes from 0xbffffb84 to 0xbffffb87

Arrays in Memory

The elements of an array will be stored in ***consecutive*** memory locations.

```
int a[5];
int i = 0;
while (i < 5) {
    printf("address of a[%d] is %p\n", i, &a[i]);
}
// prints:
// address of a[0] is 0xbffffb60
// address of a[1] is 0xbffffb64
// address of a[2] is 0xbffffb68
// address of a[3] is 0xbffffb6c
// address of a[4] is 0xbffffb70
```

Note: to print the address in printf, use **%p**.

Pointers

A pointer is a data type whose value is a reference to another variable.

```
int *ip; // pointer to int
char *cp; // pointer to char
double *fp; // pointer to double
```

In most C implementations, pointers store the memory address of the variable they refer to; i.e. they point to the variable, whose address they store.

The **&** (***address-of***) operator returns a reference to a variable.
The ***** (***dereference*** - 'de-reference' - go to the reference to get the value there) operator accesses the variable referred to by the pointer.

Pointer syntax:
[type] *[some_name] = &[something];
For example:

```
int *my_pointer = &my_variable;
```

Importantly: the **value** of the pointer is the **address** of the variable it points to

For example:

```
int i = 7;
int *ip = &i;
printf("%d\n", *ip); // prints 7
*ip = *ip * 6;
printf("%d\n", i); // prints 42
i = 24;
printf("%d\n", *ip); // prints 24
```

Like other variables, pointers need to be initialised before they are used. It is best if novice programmers initialise pointers as soon as they are declared. The value **NULL** can be assigned to a pointer to indicate it does not refer to anything. **NULL** is a #define in stdio.h and **NULL** and 0 are interchangeable (in modern C), however most programmers prefer **NULL** for readability.

Size of Pointers

Just like any other variable of a certain type, a variable that is a **pointer** also occupies space in memory. The number of bytes depends on the computer's architecture.

- 32-bit platform: pointers are likely to be 4 bytes
- 64-bit platform: pointers are likely to be 8 bytes
- Tiny embedded CPU: pointers could be 2 bytes (e.g. your microwave)

Pointer Arguments

When we pass primitive variable types as arguments to functions, they are ***passed by value*** and any changed made to them are not reflected in the caller function. Recall that scanf is a function, that takes a variable from the main function as an argument. How does a function like **scanf** manage to update the value of a variable found in the main function? It takes **pointers** to those variables as arguments!!

We use pointers to pass variables ***by reference***. By passing the address of a variable rather than its variable, we can change the value of that variable and have these changes be reflected in the caller function.

```
int main(void) {
    int i = 1;
    increment(&i);
    printf("%d\n", i);
    //prints 2

    return 0;
}

void increment(int *n) {
    *n = *n + 1;
}
```

The function `increment` is called with the address of ***i*** , and the memory of the address is passed to the variable ****n***. The function increases the value of the location referenced by ***n*** by 1.

In a sense, pointer arguments allow a function to 'return' more than one value. This increases the versatility of function. For examples, **scanf** is able to read multiple values and it uses its return value as an error status.

You have to be extremely careful when returning pointers. Returning a pointer to a local variable is **illegal** - that variable is destroyed when the function returns. But, you can return a pointer that was given as an argument.

```
int increment(int *n) {
    *n = *n + 1;
    return n;
}
```

Nested calling of functions is now possible: `increment(increment(&i));`

Array Representation

A C array has a very simple underlying representation, it is stored in an unbroken memory block and a pointer is kept at the beginning of the block.

```
char s[] = "Hi!";
printf("s: %p *s: %c\n\n", s, *s);
printf("&s[0]: %p s[0]: %c\n", &s[0], s[0]);
printf("&s[1]: %p s[1]: %c\n", &s[1], s[1]);
printf("&s[2]: %p s[2]: %c\n", &s[2], s[2]);
printf("&s[3]: %p s[3]: %c\n", &s[3], s[3]);
// prints
// s: 0x7fff4b741060 *s: H
// &s[0]: 0x7fff4b741060 s[0]: H
// &s[1]: 0x7fff4b741061 s[1]: i
// &s[2]: 0x7fff4b741062 s[2]: !
// &s[3]: 0x7fff4b741063 s[3]:
```

Since array variables are pointers, it now becomes clear why we can pass arrays to scanf without the need for address-of(&) and why arrays are passed to functions by reference.

A good explanation about pointers and arrays:
https://edstem.org/courses/1950/discussion/80387

We can even use another pointer to act as the array name.

```
int nums[] = {1, 2, 3, 4, 5};
int *p = nums;
printf("%d\n", nums[2]);
printf("%d\n", p[2]);
// both print: 3
```

Since nums acts as a pointer we can directly assign its value to the pointer p.

We can even make a pointer point to the middle of an array:

```
int nums[] = {1, 2, 3, 4, 5};
int *p = &nums[2];
printf("%d %d\n", *p, p[0]);
```

So what is the difference between an array variable and a pointer?

```
int i = 5;
p = &i; // this is OK
nums = &i; // this is an error
```

Unlike a regular pointer, an array variable is defined to point to the beginning of the array, it is constant and may not be modified

Pointer Comparison

Pointers can be tested for equality or relative order.

```
double ff[] = {1.1, 1.2, 1.3, 1.4, 1.5, 1.6};
double *fp1 = ff;
double *fp2 = &ff[0];
double *fp3 = &ff[4];
printf("%d %d\n", (fp1 > fp3), (fp1 == fp2));
// prints: 0 1
```

Note that we are comparing the values of the pointers, i.e., memory addresses, not the values the pointers are pointing to

Pointer Summary

Pointers:

- are a compound type
- usually implemented with memory addresses
- are manipulated using address-of(&) and dereference()
- should be initialised when declared
- can be initialised to NULL
- should not be dereferenced if invalid
- are used to pass arguments by reference
- are used to represent arrays
- should not be returned from functions if they point to local variables

Passing by Reference / Value

Tuesday, 10 April 2018 2:35 PM

In programs, when a function is called, variables are passed as a value
However, **if the variable is an array**, this variable will be passed as a reference.

Why?
It is complicated for a computer to replicate an array of values, but it is simple for a single value to be passed in the memory.

```
&var - passes the memory location of var
*var - accesses the value of at the address of var (aka dereference)
var - the address
```

We can get the memory address of a variable by prepending the ampersand (&) symbol.
To get the actual value of a memory address, we can prepend the asterisk (*) symbol.

This is useful for when we wish to manipulate variables in our $main$ function with another function

Consider the code below

```
#include <stdio.h>

void addOne(int *number);
void badAddOne(int number);

int main(void) {
    int number = 0;

    printf("number = %d\n", number);

    badAddOne(number);
    printf("number = %d\n", number);

    addOne(&number);
    printf("number = %d\n", number);
}

void addOne(int *n) {
    printf("\nfunction addOne called\n");
    *n += 1;
}

void badAddOne(int n) {
    printf("\nfunction badAddOne called\n");
    n += 1;
}
```

Our output:
number = 0
function badAddOne called
number = 0
function addOne called
number = 1

Look at the function for badAddOne.

The function is called with the argument $number$ (which equals to 0)
The value of $number = 0$ is copied and used as the variable n. $(n = 0)$
The function increases the value of n by one $(n = 1)$
However this function did not affect the value of $number$, which is still equal to zero.

Look at the function for addOne.

The function is called with the memory address of the variable $number$.
The memory address of $number$ is passed into the variable n.
The function increases the value of the location referenced by n by one $(number = 1)$

Extra C Features

Tuesday, 17 April 2018 9:06 PM

C Features we don't want you to use but are telling you what they are and how to use them

global variables

Variables declared outside of any function are available to all functions. They are called *external* variables or *global* variables.

```
int g = 12;

void f(void) {
    printf("The value of g is %d\n", g); // prints 12
    g = 42;
}

int main(void) {
    f();
    printf("The value of g is %d\n", g); // prints 42
    return 0;
}
```

Avoid global variables - they are **NOT** needed in COMP1511.
They make concurrency (threads) problematic and create hidden dependencies between parts of a program. They also make code harder to reuse and pollute the namespace - create a valid name everywhere you might accidentally use them. Global variables generally reduce readability although they can be useful for "meta"-purposes. e.g. turning on-off debug logging through your program.

static functions

Functions are shared between files by default. This is undesirable in large programs because name clashes become more likely. Name clashes also make code difficult to reuse.

The keyword **static** makes functions visible only within the file. In other words, **static** limits the function's scope. If a function doesn't need to be visible declare it static e.g.

```
static double helper_function(int x, double y);
```

It allows files to be *de facto* modules in C. Similarly **static** makes global variables visible only within the file. Beware, **static** has different meanings for local (function) variables.

When a function is called, its variables are created. When a function returns, its variables are destroyed. **static** changes the *lifetime* of a function's (local) variable. The value is preserved between function calls. Static variables make concurrency difficult and it also makes programs harder to read and understand. There is rarely a good reason to use static variables - so do **NOT** use them in COMP1511. Note: there is very different meaning to using **static** outside functions; poor language design much.

For example, here is a function that counts how many times it has been called.

```
void count(void) {
static int call_count = 0;
call_count++;
printf("I have been called %d times\n", call_count);
}
```

More C Operators

C provides some additional operators, which allow shorted statements. This can make your code a little more readable or a lot less readable.

- Pre/post-increment: `++i`, `i++` is the same as `i = i + 1`
- Pre/post-decrement: `--i`, `i--` is the same as `i = i - 1`
- Compound assignment operators
  - `a += b` same as `a = a + b`
  - `a -= 5` same as `a = a - 5`
  - `a *= -10` same as `a = a * -10`
  - `a /= 2` same as `a = a / 2`
  - `a %= b` same as `a = a % b`

++ and -- can be used in expression, but this is **NOT** recommended in COMP1511.

They can be used *after* the variable:

```
k = 7;
n = k--; // assign k to n, then decrement k by 1
```

```
printf("%d %d", k, n) // k=6, n=7
```

They can be used *before* the variable:

```
k = 7;
n = --k; // decrement k by 1, then assign k to n
printf("%d %d", k, n) // k=6, n=6
```

The *for* loop

There is a construct called a `for` loop:

```
for (expr1; expr2; expr3) {
    statements;
}
```

- *expr1* is evaluated before the loop starts
- *expr2* is evaluated at the beginning of each loop; if it is non-zero, the loop is repeated
- *expr3* is evaluated at the end of each loop

Example of for loop:

```
for (x = 1; x <= 10; x++) {
    printf("%d\n", x * x);
}
```

We can declare a variable if it is used only within the loop:

```
for (int x = 1; x <= 10; x++) {
    printf("%d\n", x * x);
}
```

For loops and **while** loops are equivalent. Any of the 3 expressions in the **for** loop can be omitted, but the ';' must still be present. For example:

```
printf("Enter starting number for Countdown: ");
scanf("%d", &n); // initial value entered by user
for (; n >= 0; n--) {
    printf("%d\n", n );
}
printf("Blast Off!\verb|\n|");
```

Although **NOT recommended**, the comma operator ',' can be used to squeeze multiple statements into *expr1* and *expr3*. For example:

```
for (int x=0, y=2; x < MAX; x++, y++) {
    ...
}
```

break and *continue*

break causes a loop to terminate; no more iterations are performed, and execution moves to whatever comes after the loop.
continue causes the *current* iteration of the loop to terminate, and execution moves to the next iteration.

- With **while** and **do** loops, the conditional expression is tested before moving to the next iteration
- With **for** loops, *expr3* is execute, the *expr2* is tested before moving to the next iteration

break and **continue** used sparingly can make code more readable. However, the overuse of **break** and **continue** can make code incomprehensible.

Here is a typical use of **break**:

```
for (int i = 0; i < LIMIT; i++) {
    // lots of complex things happens here
    if (/* need to stop loop immediately */ ) {
        break; // exit loop immediately
    }
    // lots more complex things happens here
}
```

Here is a typical use of **continue**:

```
for (int i = 0; i < LIMIT; i++) {
    // lots of complex things happens here
    if (/* this is not what is wanted */ ) {
        continue; // go to next loop iteration
    }
    // lots more complex things happens here
}
```

Exiting a Program

In main, **return** will terminate a program.
stdlib.h provides a function useful outside main:

```
void exit(int status);
```

status is passes to exit the same a return value of main.
stdlib.h defines **EXIT_SUCCESS** and **EXIT_FAILURE.** An **EXIT_SUCCESS** program means the program execute successfully, while and **EXIT_FAILURE** program means that the program stopped due to an error. **EXIT_SUCCESS** == 0 on Unix like and almost all other systems.

Implicit Type Conversions

Recall that C supports 'hybrid' arithmetic operations involving certain types, in a way that mirrors our expectations. For example `3 + 5.8`
An integer is added to a double, giving a double result. However at the machine level, floating point addition requires two double arguments and is a distinct operation from integer addition.

Implicit Conversions

The compiler steps in and performs and automatic conversion known as cast, from integer to double.

```
double d = 3; // 3 is converted to double
int i = 5;
```

Explicit Type Conversions

C allows us to perform our own, explicit type casts, using the syntax (*type*). For example:

```
double d1 = 1 / 2;
double d2 = 1 / (double) 2;
// d1 = 0.000000
// d2 = 0.500000
```

The values of d1 and d2 will be different.

It is good programming style to identify potentially unsafe implicit conversion types and make them explicit.

```
#include <limits.h>
#include <assert.h>
```

The compiler steps in and performs and automatic conversion known as cast, from integer to double.

```
double d = 3; // 3 is converted to double
int i = 5;
d = d + i; // i is converted to double
```

Implicit conversions are generally performed when considered 'safe'. e.g. numeric types are converted to other numeric types with larger capacity. But sometimes unsafe implicit conversions are also performed. This is an aspect of C that is often criticised. Consider:

```
int i = 1000;
char c1 = 100; // statically checked, OK
char c2 = 1000; // statically checked, warning
char c3 = i; // no warning
```

You should be mindful of implicit conversions. Often they make coding easier, but sometimes they mask programming errors.

implicit conversion types and make them explicit.

```
#include <limits.h>
#include <assert.h>
...
assert(i >= CHAR_MIN && i <= CHAR_MAX);
char c = (char) i; // for some int i
```

Note: When using explicit casts the compiler will often assume that you know what you are doing and not issue warnings even when a cast is very likely to be unsafe. For example:

```
int i = 1000;
char c = (char) i;
int *ip = (int *) i;
int nums[] = {0};
printf("%c\n", (char) i);
printf("%s\n", (char *) &i);
printf("%s\n", (char *) nums);
```

Here the casts are used to view one type as another. This is often dangerous!!!

typedef

We use the keyword `typedef` to give a name to a type:

```
typedef double real;
```

This means variables can be declared as numeric (`real`), but they will be actually be of type **double.** Do not overuse `typedef` - it can make programs harder to read. For example:

```
typedef int andrew;
andrew main(void) {
    andrew i,j;
    …
```

Using `typedef` to make programs portable:
Suppose we have a program that does floating-point calculations. If we use a typedef'ed name for all variables e.g.

```
typedef double real;

real matrix[1000][1000][1000];

real my_atanh(real x) {
    real u = (1.0 - x)/(1.0 + x);
    return -0.5 * log(u);
}
```

If we move to a platform with little RAM, we can save memory (and lose precision) by changing the typedef:

```
typedef float real;
```

structs

We have seen simple types: **int**, **char**, **double**. Variables of these types hold single values.
We have seen a compound type: **array**. Array variables hold multiple values. They are homogenous - as in every element in the array is of the same type - and they elements are selected using integer index. Array sizes can be determined at runtime.

We will now introduce another compound type: `structs`
structs hold multiple values (fields). They are heterogenous - as in the fields can be of different types . **struct** fields are selected using their names and their fields are fixed.

Example of a **struct**:
If we define a **struct** that holds COMP1511 student details:

```
#define MAX_NAME 64
#define N_LABS 10
struct student {
    int zid;
    char name[MAX_NAME];
    double lab_marks[N_LABS]
    double assignment1_mark;
    double assignment2_mark;
```

Assigning structs

Unlike arrays, it is possible to copy all components of a structure in a single assignment:

```
struct student_details student1, student2;
...
student2 = student1;
```

It is not possible to compare all components within a single comparison:

```
if (student1 == student2) // NOT allowed!
```

If you want to compare two structures, you would need to write a function to compare them component-by-component and decide whether they are the "same".

structs and functions

A structure can be passed as a parameter to a function:

```
void print_student(student_t student) {
    printf("%s z%d\n", d.name, d.zid);
}
```

Unlike arrays, a copy will be made of the entire structure, and only this copy will be passed to the function. Unlike a function a function can

```
    double lab_marks[N_LABS]
    double assignment1_mark;
    double assignment2_mark;
}; // need semi-colon to indicate you are done
// declaring the struct
```

We can declare an array to hold the details of all students:

```
struct student comp1511_students[900];
```

Note: You can't just assign strings to structs. E.g

```
comp1511_students[0].name = "Andrew";
```

You need to use **strcpy:**

```
strcpy(comp1511_students[0].name, "Andrew");
```

Combining structs and typedef

A common use of **typedef** is to give a name to a struct type.

```
struct student {
    int zid;
    char name[64];
    double lab_marks[N_LABS];
    double assignment1_mark;
    double assignment2_mark;
};

typedef struct student student_t;

student_t comp1511_students[900];
```

Programmers often use convention to separate type names.
e.g. **_t** suffix

Nested Structures

One structure can be nested inside another:

```
typedef struct date     Date;
typedef struct time     Time;
typedef struct speeding Speeding;

struct date {
    int day, month, year;
};
struct time {
    int hour, minute;
};
struct speeding {
    Date date;
    Time time;
    double speed;
    char plate[MAX_PLATE];
};
```

Unlike arrays, a copy will be made of the entire structure, and only this copy will be passed to the function. Unlike a function a function can return a struct:

```
student_t read_student_from_file(char filename[]) {
    ....
}
```

Pointers to structs

If a function needs to modify a structs field or if we want to avoid the inefficiency of copying the entire struct, we can instead pass a pointer to the struct as a parameter:

```
int scan_zid(student *s) {
    return scanf("%d", &((*s).zid));
}
```

The "arrow" operator is more readable:

```
int scan_zid(student *s) {
    return scanf("%d", &(s->zid));
}
```

If **s** is a pointer to a struct, **s->field** is equivalent to **(*s).field**

Malloc

Wednesday, 2 May 2018 2:08 PM

malloc and free

For example, let's assume we need a block of memory to hold a string of say 100 000 000 ints

```
int *p;
p = malloc(100000000 * sizeof (int));
if(p == NULL) {
    fprintf(stderr, "Error: could not allocate memory!\n");
    exit(1); // means something went wrong
}

// we can now use the pointer
// … lots of things to do

free(p); // free up the memory that was used
```

malloc() allocates memory in "the heap" and returns a pointer to a block of memory (it returns the address of the memory it is allocated). **malloc()** returns a (void *) pointer and can be assigned to any pointer type. If insufficient memory is available then **malloc()** returns **NULL.**

Memory "lives" on forever until we free it. **free()** indicates that you have finished using a block of memory. Continuing to use memory after memory after **free()** results in in very nasty bugs. Using **free()** on a memory block twice can also cause bad bugs. If a program keeps calling **malloc()** without corresponding **free()** calls, then the program's memory will grow steadily larger. This is called a **memory leak**. Memory leaks are major issues for long running programs.

Newton's 3rd Law of Memory Management:
'For every malloc, there is an equal and opposite free"
Why? Because memory is a finite resource.

sizeof

sizeof is a C operator that yields bytes when needed for a type or variable.
You use it like this:

```
sizeof (type)
sizeof variable_name
```

Note: unusual syntax (badly designed) brackets indicate argument is a type
You should use **sizeof** for every malloc call.
Here are some more examples of using **sizeof**:

```
printf("%ld", sizeof (char)); // 1
printf("%ld", sizeof (int)); // 4 commonly
printf("%ld", sizeof (double)); // 8 commonly
printf("%ld", sizeof (int[10])); // 40 commonly
printf("%ld", sizeof (int *)); // 4 or 8 commonly
printf("%ld", sizeof "hello"); // 6
```

Syntax = rules of the language
Semantics = meaning behind the language

Linked Lists

Wednesday, 2 May 2018 2:08 PM

Self-Referential Structures

We can define a structure containing a pointer to the same type of structure like this:

```
struct node {
    struct node *next;
    int          data;
};
```

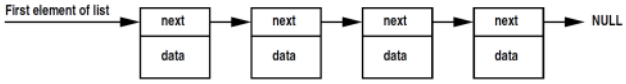
These "self-referential" pointers can be used to build larger "dynamic" data structures out of smaller building blocks.

Linked Lists

The most fundamental of these dynamic data structures it the **Linked List.**

It is based on the idea of a sequence of data items or nodes. Linked lists are more flexible than arrays:

- Items do not have to be located next to each other in memory
- Items can easily rearranged by altering pointers
- The number of items can change dynamically
- Items can be added or removed in any order



A **linked list** is a sequence of items.
Each item contains data and a pointer to the next item. You need to separately store a pointer to the first item or "head" of the list. The last item in the list is special - it contains NULL in its next field instead of a pointer to an item .

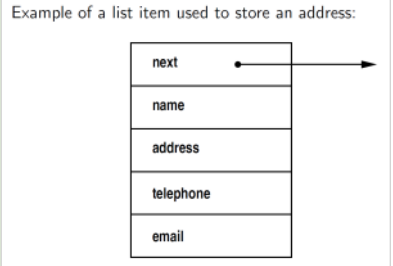
Example of List Item

Example of a list item used to store an address

In C code:

```
struct address_node {
    struct address_node *next;
    char *name;
    char *address;
    char *telephone;
    char *email;
};
```

In a diagram:



List Items

List items may hold large amounts of data or many fields. For simplicity, we'll assume each list item only needs to store a single int.

```
struct node {
    struct node *next;
    int data;
};
```

List Operations

Some basic lists operations are:

- Creating a new item with specified data
- Searching for an item with particular data
- Inserting a new item to the list
- Removing an item from the list

Many other operations are possible.

Here are various codes showing what you can do with **linked lists**:

Creating a list item

```
// Create a new struct node containing the specified data,
// and next fields, return a pointer to the new struct node.

struct node *create_node(int data, struct node *next) {
    struct node *n;
    n = malloc(sizeof (struct node));
    if (n == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
    n->data = data;
    n->next = next;
    return n;
}
```

Building a list

```
// Building a list containing the 4 ints: 13, 17, 42, 5

struct node *head = create_node(5, NULL);
head = create_node(42, head);
head = create_node(17, head);
head = create_node(13, head);
```

Summing a list

Finding an item in a list: using a shorter **while** loop
Same function but using a more concise while loop.
Shorter does not always mean more readable.

```
// return pointer to first node containing
// specified value, return NULL if no such node

struct node *find_node(struct node *head, int data) {
    struct node *n = head;
    while (n != NULL && n->data != data) {
        n = n->next;
    }
    return n;
}
```

Finding an item in a list: recursive
Same function but function calls itself.

```
// return pointer to first node containing
// specified value, return NULL if no such node

struct node *find_node(struct node *head, int data) {
    if (head == NULL) {
        return NULL;
    }
    if (head->data == data) {
        return head;
    }
    return find_node(head->next, data);
}
```

Finding an item in a list: shorter recursive
Same function but a more concise recursive version.
Shorter does not always mean more readable.

```
// return pointer to first node containing
// specified value, return NULL if no such node

struct node *find_node(struct node *head, int data) {
    if (head == NULL || head->data == data) {
        return head;
    }
    return find_node(head->next, data);
}
```

Printing a list: in python syntax

```
// print contents of list in Python syntax
void print_list(struct node *head) {
    printf("[");
        for (struct node *n = head; n != NULL; n = n->next) {
        printf("%d", n->data);
            if (n->next != NULL) {
                printf(", ");
            }
        }
    printf("]");
}
```

Finding the last item in a list

```
// return pointer to last node in list
// NULL is returned if list is empty

struct node *last(struct node *head) {
    if (head == NULL) {
        return NULL;
    }
    struct node *n = head;
    while (n->next != NULL) {
        n = n->next;
    }
    return n;
}
```

Appending to a list

```
// append integer to end of list
struct node *append(int value, struct node *head) {
    struct node *n;
    n = create_node(value, NULL);
    if (head == NULL) {
        // new node is now head of the list
        return n;
    } else {
        struct node *l = last(head);
        l->next = n;
        return list;
```

```
head = create_node(17, head);
head = create_node(13, head);
```

Summing a list

```
// return sum of list data fields

int sum(struct node *head) {
    int sum = 0;
    struct node *n = head;
    // execute until end of list
    while (n != NULL) {
        sum += n->data;
        // make n point to next item
        n = n->next;
    }
    return sum;
}
```

Summing a list: using a **for** loop

```
// return sum of list data fields

int sum(struct node *head) {
    int sum = 0;
    for (struct node *n = head; n != NULL; n = n->next) {
        sum += n->data;
    }
    return sum;
}
```

Summing a list: recursive
Same function but using a recursive call.

```
// return sum of list data fields

int sum2(struct node *head) {
    if (head == NULL) {
        return 0;
    }
    return head->data + sum2(head->next);
}
```

Finding an item in a list

```
// return pointer to first node containing
// specified value, return NULL if no such node

struct node *find_node(struct node *head, int data) {
    struct node *n = head;
    // search until end of list reached
        while (n != NULL) {
            if (n->data == data) {
                // matching item found
                return n;
            }
        // make n point to next item
        n = n->next;
        }
    // item not in list
    return NULL;
}
```

Finding an item in a list: using a **for** loop

```
// return pointer to first node containing
// specified value, return NULL if no such node

struct node *find_node(struct node *head, int data) {
    for (struct node *n = head; n != NULL; n = n->next) {
        if (n->data == data) {
            return n;
        }
    }
    return NULL;
}
```

```
    } else {
        struct node *l = last(head);
        l->next = n;
        return list;
    }
}
```

Deleting all items from a list

```
// Delete all the items from a linked list.

void delete_all(struct node *head) {
    struct node *n = head;
    struct node *tmp;
    while (n != NULL) {
        tmp = n;
        n = n->next;
        free(tmp);
    }
}
```

Inserting a node into an ordered list

```
struct node *insert(struct node *head, struct node *node) {
    struct node *previous;
    struct node *n = head;
    // find correct position
    while (n != NULL && node->data > n->data) {
        previous = n;
        n = n->next;
    }
    // link new node into list
    if (previous == NULL) {
        head = node;
    } else {
        previous->next = node;
    }
    node->next = n;
    return head;
}
```

Inserting a node into an ordered list: recursive

```
struct node *insert(struct node *head, struct node *node) {
    if (head == NULL || head->data >= node->data) {
        node->next = head;
        return node;
    }
    head->next = insert(head->next, node);
    return head;
}
```

Deleting a node from a list

```
struct node *delete(struct node *head, struct node *node) {
    if (node == head) {
        head = head->next; // remove first item
        free(node);
    } else {
        struct node *previous = head;
        while (previous != NULL && previous->next != node) {
            previous = previous->next;
        }
        if (previous != NULL) { // node found in list
            previous->next = node->next;
            free(node);
        } else {
            fprintf(stderr, "warning: node not in list\n");
        }
    }
    return head;
}
```

Deleting a node from a list: recursive

```
struct node *delete(struct node *head, struct node *node) {
    if (head == NULL) {
        fprintf(stderr, "warning: node not in list\n");
    } else if (node == head) {
        head = head->next; // remove first item
        free(node);
    } else if (head == head) {
        head->next = delete(head->, node)
    }
    return head;
}
```

Multiple C files, Header files

Tuesday, 15 May 2018 1:17 PM

We can make programs with multiple .c files. It is no different from making/compiling a program from one .c file.

```
dcc -o program_name first.c second.c
```

Note: You cannot have more than one main function. So make sure there is only **ONE** main function amongst the code in your .c files

Using functions from another file

first.c

```
void hello(void) {
    printf("Hello!\n");
}
int square(int n) {
    return n*n;
}
```

second.c

```
#include <stdio.h>
#include "first.h"

int main(void) {
    hello();
    printf("%d", square(5));
}
```

first.h

```
#ifndef FIRST_H
#define FIRST_H

void hello(void);
int square(int n);

#endif
```

Notes:

- In second.c , we use for **#include "first.h"** use quotation marks because it is not a library but something I made
- **#ifndef** is a header guard. This avoids errors if the .h file is included multiple times
- Don't include the .h file in the command line.
- When you has define a .h file, it needs to be in the same folder as the .c files using it.

Stacks and Queues

Wednesday, 16 May 2018 2:24 PM

Stacks and Queues are ubiquitous data structures in computing. They are part of many algorithms and are a good example of abstract data types.

A type is **concrete** if a user of that type has knowledge of how it works.
A type is **abstract** if a user has no knowledge of how it works.

Stack - Abstract Data Type

A ***stack*** is a collection of items such that the ***last*** item to enter is the ***first*** one to exit (LIFO - Last In, First Out). They are based on the idea of stacks of books, or plates.
Essential ***stack*** operations include:

- `push()` - add new item to stack
- `pop()` - remove top item from stack

Additional ***stack*** operations are:

- `top()` - fetch top item (but don't remove it)
- `size()` - no. of items
- `is_empty()` - check if stack is empty

Queue - Abstract Data Type

A ***queue*** is a collection of items such that the ***first*** item to enter is the ***first*** one to exit (FIFO - First In, First Out). It is based on the idea of queueing at a bank or shop. Essential ***queue*** operations include:

- `enqueue()` - add new item to queue
- `dequeue()` - remove front item from queue

Additional ***queue*** operations are:

- `front()` - fetch the front item (but don't remove it)
- `size()` - no. of items
- `is_empty()` - check is queue is empty

Stack Applications

Stacks are used for:

- Page-visited history in a web browser
- Undoing a sequence in a text editor
- Checking for balanced brackets
- HTML tag matching
- Postfix (RPN) calculator
- Chain of function calls in a program

Queue Applications

Queues are used for:

- Waiting lists, bureaucracy
- Access to shared resource (printers, etc.)
- Phone call centres
- Multiple processors in a computer

Stack - Abstract Data Type - C Interface

```
typedef struct stack_internals *stack;
stack stack_create(void);
void stack_free(stack stack);
void stack_push(stack stack, int item);
int stack_pop(stack stack);
int stack_is_empty(stack stack);
int stack_top(stack stack);
int stack_size(stack stack);
```

Using stacks in C interface:

```
stack s;
s = stack_create();
stack_push(s, 10);
stack_push(s, 11);
stack_push(s, 12);
printf("%d\n", stack_size(s)); // prints 3
printf("%d\n", stack_top(s));  // prints 12
printf("%d\n", stack_pop(s));  // prints 12
printf("%d\n", stack_pop(s));  // prints 11
printf("%d\n", stack_pop(s));  // prints 10
```

The implementation of a stack is **opaque** (hidden from the user). Users program cannot depend on how stack is implemented. Stack implementation can change without risk of breaking users program. This type of **information hiding** is crucial to managing complexity in large software systems.

Queue - Abstract Data Type - C Interface

```
typedef struct queue_internals *queue;
queue queue_create(void);
void queue_free(queue queue);
void queue_enqueue(queue queue, int item);
int queue_dequeue(queue queue);
int queue_is_empty(queue queue);
int queue_front(queue queue);
int queue_size(queue queue);
```

Using queues in C interface:

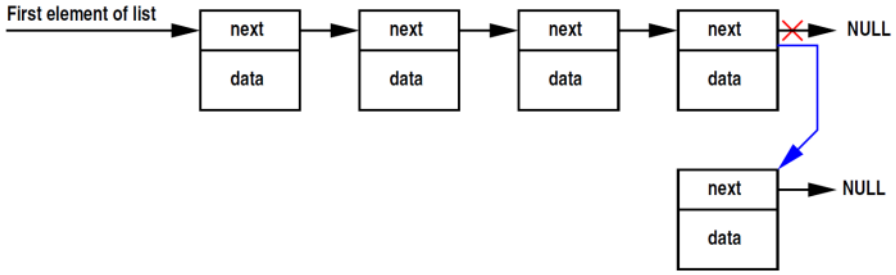
```
queue q;
q = queue_create();
queue_enqueue(q, 10);
queue_enqueue(q, 11);
queue_enqueue(q, 12);
printf("%d\n", queue_size(q));    // prints 3
printf("%d\n", queue_front(q));   // prints 10
printf("%d\n", queue_dequeue(q)); // prints 10
printf("%d\n", queue_dequeue(q)); // prints 11
printf("%d\n", queue_dequeue(q)); // prints 12
```

Like stacks the implementation of queues is **opaque**. Queue implementation can change without risk of breaking user programs.

Implementing A Stack with a Linked List

A stack can be implemented using a linked list, by adding and removing at the head [`push()` and `pop()`]. For a queue, we need to either add or remove at the tail.

Adding an item to the Tail of a list:



Adding an item to the tail is achieved by making the last node of the list point to the new node. We first need to scan along the list to find the last item.

```
struct node *add_to_tail( *new_node, struct node *head) {
     if (head == NULL) { // list is empty
          head = new_node;
     } else { // list not empty
          struct node *node = head;
          while (node->next != NULL) {
               node = node->next; // scan to end
          }
          node->next = new_node;
     }
     return head;
}
```

Efficiency Issues:
Unfortunately, this implementation is very slow. Every time a new item is inserted, we need to traverse the entire list (which could be very large). We can do the job more efficiently if we retain a direct link to the last item of "tail" of the list

```
if (tail == NULL) { // list is empty
     head = node;
} else { // list not empty
     tail->next = node;
}
tail = node;
```

Note: There is no way to efficiently *remove* items from the tail

Postfix Calculator Example

Some early calculators and programming languages used convention known as *Reverse Polish Notation* (RPN), where the operator comes after the two operands rather than between them:
1 2 +
result = 3
3 2 *
result = 6
4 3 + 6 *
result = 42
1 2 3 4 + * +
result = 15

A calculator using RPN is called a *Postfix Calculator*, it can be implemented using a stack:

- When a number is entered: push it onto the stack
- When an operator is entered: pop the two items from the stack, apply the operator to them, and push the result back onto the stack

```
#include <stdio.h>
#include <ctype.h>
#include "stack.h"
int main(void) {
     int ch;
     stack s = stack_create();
     while ((ch = getc(stdin)) != EOF) {
          if (ch == '\n') {
               printf("Result: %d\n", stack_pop(s));
          } else if (isdigit(ch)) {
```

```
            ungetc(ch, stdin); // put first digit back
            int num;
            scanf("%d", &num); // now scan entire number
            stack_push(s, num);
        } else if (ch == '+' || ch == '-' || ch == '*') {
            int a = stack_pop(s);
            int b = stack_pop(s);
            int result;
            if (ch == '+') {
                result = b + a;
            } else if (ch == '-') {
                result = b - a;
            } else {
                result = b * a;
            }
            stack_push(s, result);
        }
    }
}
```

Illegal C

Wednesday, 23 May 2018 2:33 PM

Consequences of bugs

- A compilers gives syntax/semantic errors *(if you're very lucky)*
- The program halts with run-time error *(if you're lucky)*
- The program never halts *(if you're lucky-ish)*
- The program halts, but with incorrect results *(if you're unlucky)*
- The program appears correct, but has security holes *(if you're unlucky)*

Changed Variable

```
int a[10];
int b[10];
printf("a[0] is at address %p\n",&a[0]);
printf("a[9] is at address %p\n", &a[9]);
printf("b[0] is at address %p\n",&b[0]);
printf("b[9] is at address %p\n", &b[9]);
for (int i = 0; i < 10; i++) {
    a[i] = 77;
}
for (int i = 0; i <= 12; i++) {
    b[i] = 42;
}
for (int i = 0; i < 10; i++) {
    printf("%d ", a[i]);
}
printf("\n");
```

The C program assigns to b[10]..b[12], which do not exist.
The consequence could be anything - a C implementation is permitted to behave in any manner given an invalid program. On gcc 6.3 on Linux/x86 64 it happens to change b[0] to 42:

```
$ gcc invalid_array_index0.c
$ a.out
a[0] is at address 0x7fffc9cbcbf0
a[9] is at address 0x7fffc9cbcc14
b[0] is at address 0x7fffc9cbcbc0
b[9] is at address 0x7fffc9cbcbe4
42 77 77 77 77 77 77 77 77 77
```

Changed Termination

```
int i;
int a[10];
printf("i is at address %p\n", &i);
printf("a[0] is at address %p\n", &a[0]);
printf("a[9] is at address %p\n", &a[9]);
printf("a[11] would be stored at address %p\n", &a[10]);
for (i = 0; i <= 11; i++) {
    a[i] = 0;
}
```

Another invalid C program assigning to a non-existent array element.
On gcc 6.3 on Linux/x86 64 it happens to assigns to i and the loop doesn't terminate.
So a character error makes the program invalid, and seemingly certain termination does not occur.

```
$ gcc invalid1.c
```

```
$ a.out
i is at address 0x7fffbb72bfdc
a[0] is at address 0x7fffbb72bfb0
a[9] is at address 0x7fffbb72bfd4
a[10] is equivalent to address 0x7fffbb72bfd8
....
```

Changed Variable in Another Function

```
int main(void) {
     int answer = 36;
     f(5);
     printf("answer=%d\n", answer); // prints 42
     return 0;
}
void f(int x) {
     int a[10];
     // a[19] doesn't exist
     // with gcc 6.3 on Linux/x86_64 variable answer
     // in main happens to be where a[19] would be
     a[19] = 42;
}
```

Yet another invalid C program assigning to a non-existent array element. On gcc 6.3 on Linux/x86 64 it changes the variable answer in the calling function main.

```
$ gcc invalid2.c
$ a.out
answer=42
```

Changed Function Return Location

```
void f() {
     int a[10];
     // on gcc-6.3/Linux x86
     // change function's return address on stack
     // causing function to return after the line
     // where answer is assigned 24
     a[14] += 7;
}

int main(void) {
     int answer = 42;
     f();
     answer = 24;
     printf("answer=%d\n", answer);
     return 0;
}
```

Yet another invalid C program assigning to a non-existent array element. With gcc 6.3 on Linux/x86 64 it changes where the function returns in main.

```
$ gcc invalid3.c
$ a.out
answer=42
```

Bypassing Authentication

```
int authenticated = 0;
char password[8];
printf("Enter your password: ");
```

```
gets(password);
if (strcmp(password, "secret") == 0) {
authenticated = 1;
}
// a password longer than 8 characters will overflow
// array password on gcc 6.3 on Linux/x86_64 this can
// overwrite the variable authenticated and allow access
if (authenticated) {
     printf("Welcome. You are authorized.\n");
} else {
     printf("Welcome. You are unauthorized. ");
     printf("Your death will now be implemented.\n");
     printf("Welcome. You will experience ");
     printf("a tingling sensation and then death. \n");
     printf("Remain calm while your life is extracted.\n");
}
```

Yet another invalid C program assigning to a non-existent array element.
A password longer than 8 characters will overflow the array password. This is often termed **buffer-overflow**.

```
$ gcc invalid4.c
$ a.out
Enter your password: secret
Welcome. You are authorized.
$ a.out
Enter your password: wrong
Welcome. You are unauthorized.
Your death will now be implemented.
Welcome. You will experience a
tingling sensation and then death.
Remain calm while your life is extracted.
$ a.out
Enter your password: longcorrectpassword
Welcome. You are authorized.
```

Implementation vs Language

C was designed for much smaller, slower computers (28K of RAM, 1mhz clock).
Program speed/size is much more important for programs than dominated language choice.
Most C implementations still focus on maximising performance of valid programs.
Most C implementations do not check array bounds or for arithmetic overflow because this have performance costs. The C definition does not entail this.
A C implementation can check array bounds and halt if invalid indexes are used.
A C implementation could check and halt if an uninitialised value is used - but is difficult/expensive to track for arrays.

Address Sanitizer extension to gcc/clang

gcc -fsantitize=address gives a very different C implementation. Invalid array indexes, pointer dereferences and some other invalid use of the string library function are detected. Performance cost - execution from 1.2-10+x slower.
Information cryptic but note source code line indicated. e.g.:

```
$ cd /home/cs1511/public_html/lec/illegal_C/code/
$ gcc -g -fsanitize=address debug_examples.c
$ ./a.out 3
ASAN:DEADLYSIGNAL
=========================
==16917==ERROR: AddressSanitizer: SEGV on unknown address
0x000000000014 (pc 0x55819087cd2c bp 0x7ffd02a40bb0 ....
```

```
    #0 0x55819087cd2b in test3 debug_examples.c:33
    #1 0x55819087d19c in main debug_examples.c:96
    #2 0x7fccf078d2b0 in __libc_start_main (/lib/...
    #3 0x55819087caf9 in _start ...
....
```

dcc uses -fsanitize=address (with clang) but makes messages more comprehensible for beginner programmers:

```
$ cd /home/cs1511/public_html/lec/illegal_C/code/
$ dcc debug_examples.c
$ ./a.out 3
ASAN:DEADLYSIGNAL

debug_examples.c:33 runtime error - illegal array, pointer
or other operation
Execution stopped in test3() in debug_examples.c line 33:

    int *a = NULL;
    // dereferencing NULL pointer
--> a[5] = 42;
}
Values when execution stopped:
```

Address Sanitizer does not detect the use of uninitialised values. e.g.:

```
% ./debug_examples 4
0
1
2
3
-2115323248
5
6
7
8
9
```

valgrind - another debugging/testing tool

valgrind works on x86 machine code - not C specific.
valgrind runs the code on a virtual machine and detects use of **uninitialised memory**.
It also picks up many invalid array indexes and pointer dereferences: Large performance penalty - and slow start time.

```
% valgrind ./debug_examples 4
==1932== Memcheck, a memory error detector
==1932== Copyright (C) 2002-2010, and GNU GPL'd, ...
==1932== Using Valgrind-3.6.1 and LibVEX; rerun ...
==1932== Command: ./debug_examples 4
==1932==
0
1
2
3
==1932== Use of uninitialised value of size 8
==1932== at 0x521AF0B: _itoa_word (_itoa.c:195)
==1932== by 0x521D3B6: vfprintf (vfprintf.c:1619)
==1932== by 0x400FBF: test4 (debug_examples.c:45)
==1932== by 0x401317: main (debug_examples.c:92)
==1932==
```

```
...
```

dcc -valgrind

dcc -valgrind causes valgrind to be used to run your program. It makes messages more comprehensible for beginner programmers:

```
$ dcc --valgrind debug_examples.c
% ./a.out 4
Runtime error: uninitialized variable accessed.
Execution stopped in test4() debug_examples.c line 45:
     // accessing uninitialized array element (a[4])
     for (i = 0; i < 10; i++)
     --> printf("%d\n", a[i]);
}

Values when execution stopped:
a = {0, 1, 2, 3, -16776544, 5, 6, 7, 8, 9}
i = 4
a[i] = -16776544
```

Searching and Sorting

Wednesday, 23 May 2018 2:33 PM

Efficiency

COMP1511 focuses on writing programs, but efficiency is also important. We often need to consider:

- Execution time
- Memory use

A **correct** but slow program can be useless. Efficiency often depends on the size of the data being processed.
Understanding this dependency lets us predict program performance on larger data.

We can find out whether a program is efficient or not through:

- ***Empirical approach*** - run the program several times with different input sizes and measure the time taken
- ***Theoretical approach*** - try to count the number of operations performed by the algorithm on input of size *n*

Searching

Linear Search Unordered Array

```
int linear_search(int array[], int length, int x) {
    for (int i = 0; i < length; i = i + 1) {
        if (array[i] == x) {
            return 1;
        }
    }
    return 0;
}
```

An informal analysis:
Operations:

- Start at the first element
- Inspect each element in turn
- Stop when you find **X** or reach the end

If there are **N** elements to search:

- Best case scenario: we only check 1 element
- Worst case scenario: we need to check **N** elements
- If the element is in the list we will check on average **N/2** elements
- If it is not in the list, it will check **N** elements

Linear Search Ordered Array

```
int linear_ordered(int array[], int length, int x)
{
    for (int i = 0; i < length; i = i + 1) {
        if (array[i] == x) {
            return 1;
        } else if (array[i] > x) {
            return 0;
        }
    }
    return 0;
}
```

An informal analysis:
Operations:

- Start at the first element
- Inspect each element in turn
- Stop when you find **X** or find a value **>X** or reach the end

If there are **N** elements to search:

- Best case scenario: we only check 1 element
- Worst case scenario: we need to check **N** elements
- If the element is in the list we will check on average **N/2** elements
- If it is not in the list, it will check **N/2** elements

Binary Search Ordered Array

```
int binary_search(int array[], int length, int x) {
    int lower = 0;
    int upper = length - 1;
    while (lower <= upper) {
        int mid = (lower + upper)/ 2;
        if (array[mid] == x) {
            return 1;
```

Sorting

Aim: to rearrange a sequence so it is in non-decreasing order
Advantages:

- Sorted sequences can be searched efficiently
- Items with equal keys are located together

Disadvantages:

- Simple obvious algorithms are too slow at sorting large sequences
- Better algorithms can sort very large sequences

Sorting has been studied extensively and many algorithms have been proposed.
One slow obvious algorithm is **bubblesort** and one fast algorithm is: **quicksort**

Bubblesort Code:

```
void bubblesort(int array[], int length) {
    int swapped = 1;
    while (swapped) {
        swapped = 0;
        for (int i = 1; i < length; i = i + 1) {
            if (array[i] < array[i - 1]) {
                int tmp = array[i];
                array[i] = array[i - 1];
                array[i - 1] = tmp;
                swapped = 1;
            }
        }
    }
}
```

Quicksort - Code

```
void quicksort(int array[], int length) {
    quicksort1(array, 0, length - 1);
}
void quicksort1(int array[], int lo, int hi) {
    if (lo >= hi) {
        return;
    }
    int p = partition(array, lo, hi);
    // sort lower part of array
    quicksort1(array, lo, p);
    // sort upper part of array
    quicksort1(array, p + 1, hi);
}

int partition(int array[], int lo, int hi) {
    int i = lo, j = hi;
    int pivotValue = array[(lo + hi) / 2];
    while (1) {
        while (array[i] < pivotValue) {
            i = i + 1;
        }
        while (array[j] > pivotValue) {
            j = j - 1;
        }
        if (i >= j) {
            return j;
        }
```

```
    while (lower <= upper) {
        int mid = (lower + upper)/ 2;
        if (array[mid] == x) {
            return 1;
        } else if (array[mid] > x) {
            upper = mid - 1;
        } else {
            lower = mid + 1;
        }
    }
    return 0;
}
```

An informal analysis:

Operations:

- Start with an entire array
- At each step half the range the element may be in
- Stop when you find **X** or when the range is empty

If there are **N** elements to search:

- Best case scenario: we only check 1 element
- Worst case scenario: we need to check $\mathbf{log_2 N + 1}$ elements
- If the element is in the list we will check on average $\mathbf{log_2 N}$ elements

$log_2(N)$ grows very slowly:

- $log_2 10 = 3.3$
- $log_2 1000 = 10$
- $log_2 1000000 = 20$
- $log_2 1000000000 = 30$
- $log_2 1000000000000 = 40$

Physicists estimate 1080 atoms in universe: $log_2(1080) = 240$

Binary search all atoms in universe in < 1 microsecond

```
        }
        if (i >= j) {
            return j;
        }
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
        i = i + 1;
        j = j - 1;
    }
    return j;
}
```

Quicksort and Bubblesort Compared

If we use quicksort and bubblesort code, we see:

Array size (n)	Bubblesort operations	Quicksort operations
10	81	24
100	8415	457
1000	981018	9351
10000	98790120	102807

- Bubblesort is proportional to **n**
- Quicksort is proportional to $\mathbf{n log_2 n}$
- If **n** is small, there is little difference between the algorithms
- If **n** is large, there is a significant difference between the algorithms
- For large **n**, you need a good sorting algorithm like quicksort