Systems Programming

Lecture 5: Data Types & Basic Functions

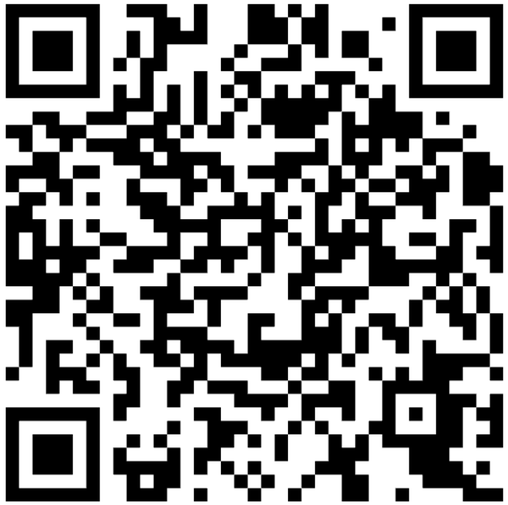
Stuart James

stuart.a.james@durham.ac.uk

Durham University

Recap

https://PollEv.com/stuartjames



Recap: Makefiles

- When we have a number of files to compile together, we need a rule-set to perform this.
  - Provided by the `make` command
- Requires a rule-file called the Makefile


Recap: Makefiles

- When we have a number of files to compile together, we need a rule-set to perform this.
  - Provided by the `make` command
- Requires a rule-file called the Makefile

In [ ]:

```
CC=gcc
CFLAGS=-I.
DEPS = counter.h sales.h

all: counter.o sales.o main.c
    (CC) -o program main.c counter.o sales.o

%.o: %.c (DEPS)
    (CC) -c -o @< $(CFLAGS)

clean:
    rm -rf program counter.o sales.o
```


Recap: Iteration statements

C provides three iteration statements:

- The `while` statement is used for loops whose controlling expression is tested before the loop body is executed - `while (a > 100) {...}`
- The `do` statement is used if the expression is tested after the body is executed

```
do {...} while (a > 100);
```

- The `for` statement is convenient for loops that increment or decrement a counting variable

```
for (a = 199; a > 100; a = a - 1) {...}
```


To ++or to++

- No, not C++ (yet)
- `x++` means `x=x+1` - We can also have `++x`, which also means `x=x+1`


To ++or to++

- No, not C++ (yet)
- `x++` means `x=x+1` - We can also have `++x`, which also means `x=x+1`

In [27]:

```c
// what is y?
#include <stdio.h>
int main(){
    int x = 5;
    int y = x++;
    printf("y is %d\n", y);
    printf("x is %d\n", x);
}
```

```
y is 5
x is 6
```


To ++or to++

- No, not C++ (yet)
- `x++` means `x=x+1` - We can also have `++x`, which also means `x=x+1`

In [27]:

```c
// what is y?
#include <stdio.h>
int main(){
    int x = 5;
    int y = x++;
    printf("y is %d\n", y);
    printf("x is %d\n", x);
}
```

```
y is 5
x is 6
```

- `x++` returns the value of `x` first, then increments
- `++x` increments first, then returns the value of `x`


To ++or to++

- And what about this?


To ++or to++

- And what about this?

In [3]:

```c
#include <stdio.h>

int main(){
    int x = 5;
    int y = x++;
    printf("y is %d\n", y);
}
```

```
y is 5
```


To ++or to++

- Can also have `--`, `+=`, `-=`, `*=` , `/=`, `%=`

```
x += 5;
```


Check your understanding

- What does this code output?


Check your understanding

- What does this code output?

In [31]:

```c
#include<stdio.h>

#define TRIPLE(a) 3*a

int main() {
  int x = 1;
  int y = 2;
  printf("%d\n",TRIPLE(y+x));
}
```

```
7
```


Check your understanding

- What does this code output?

In [31]:

```c
#include<stdio.h>

#define TRIPLE(a) 3*a

int main() {
  int x = 1;
  int y = 2;
  printf("%d\n",TRIPLE(y+x));
}
```

```
7
```

Order of precedence is important:

https://en.cppreference.com/w/c/language/operator_precedence


Check your understanding

- What does this code output?


Check your understanding

- What does this code output?

In [34]:

```c
#include<stdio.h>

int main() {
  int x = 2;
  x *= 1 + 2;
  printf("%d\n",x);
}
```

```
6
```


Check your understanding

- What does this code output?


Check your understanding

- What does this code output?

In [35]:

```c
#include<stdio.h>

int main() {
  int x = 3;
  int y = 2, z = 2;
  x = y == z;
  printf("%d\n",x);
}
```

```
1
```


Check your understanding

- What does this code output?


Check your understanding

- What does this code output?

In [36]:

```c
#include<stdio.h>

int main() {
  int x = 1;
  int y = 2, z = 0;
  x += y = z = 4;
  printf("x=%d, y=%d, z=%d\n",x,y,z);
}
```

```
x=5, y=4, z=4
```


The `switch` statement

- This has the form:

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

- Warning: if there is no `break` statement, execution falls through!
  - A "fall-through" occurs when the program continues to execute code in subsequent case blocks even after a matching case is found.


The `switch` statement

Example:


The `switch` statement

Example:

In [22]:

```c
#include <stdio.h>

int main(){
    int x = 0;
    switch(x){
      case 0:
        printf("x is 0\n");
//         break;
      case 1:
        printf("x is 1\n");
//         break;
      case 2:
        printf("x is 2\n");
//         break;
      default:
        printf("x is some other value!\n");
        // Putting a final break statement is good practice
        break;
    }
    return 0;
}
```

```
x is 0
x is 1
x is 2
x is some other value!
```


In [25]:

```c
// Fall-through can be intentional
// In this case we want to print what days are left in the week based on numerical value of day:

#include <stdio.h>

int main() {
    int day = 5;

    switch (day) {
        case 1:
            printf("Monday\n");
        case 2:
            printf("Tuesday\n");
        case 3:
            printf("Wednesday\n");
        case 4:
            printf("Thursday\n");
        case 5:
            printf("Friday\n");
            break; // separating weekdays and weekends
        case 6:
            printf("Saturday\n");
        case 7:
            printf("Sunday\n");
            break;
        default:
            printf("Invalid day\n");
    }

    return 0;
}

```

```
Friday
```

Variables

Variables and constants are the basic data objects manipulated by a program.

- **Declarations:** declare the variables used, their type and possibly initial value.
- **Expressions:** combine variables and constants to form new values.

```
int i = 6+7*3;
```


Data types

- Every C variable must have a type (strongly typed language)
  - `char`: a single byte -- often used to store a character
  - `short`: an integer type, represents small whole numbers
  - `int`: an integer type, represents whole numbers
  - `long int`, `long long int`: an integer type, represents large or very large whole numbers
  - `float`: single precision floating point number
  - `double`, `long double`: double precision floating point number
  - a few others


Data types

- Every C variable must have a type (strongly typed language).
- On 64-bit Linux systems these require 1 (char), 2 (short) ,4 (int, long, float),8 (long long, double), and 16 (long double) bytes.
- Size in bytes needed for memory management and I/O.
- Compiler can choose size of integers subject to:
  - `short int` and `int` are at least 16 bits (2 bytes)
  - `long int` is at least 32 bits (4 bytes)


Data type qualifiers

- On 64-bit Linux:
  - `char` 1 byte -128 to 127
  - `short int` 2 bytes -32768 to +32767
  - `int` 4 bytes -2147483648 to +2147483647
  - `long int` 8 bytes -9223372036854775808 to +9223372036854775807


`signed` vs `unsigned`

- `signed`/`unsigned`: applies to `char` or integer types.
- `unsigned` integers are always positive or 0
  - `signed char` 8 bits (1 byte) integer [-128,127]
  - `unsigned char` 8 bits (1 byte) integer [0,255]
- `<limits.h>` and `<float.h>` specify what limits apply on a given system
- they are system and architecture dependent


Character constants

- These are integer values that are written as a character in single quotes.
  - e.g., `'0'` = `48` in the ASCII character set
  - https://www.ascii-code.com/
- These can also include escape characters:
  - `'\n'` newline character
  - `'\a'` alert (bell) character
  - `'\t'` horizontal tab
  - `'\0'` `NULL` character


Character constants

- On UNIX, you can run the `man ascii` command for more information. (Press `q` to exit.)


Character constants

- On UNIX, you can run the `man ascii` command for more information. (Press `q` to exit.)

In [41]:

```c
#include <stdio.h>

int main(){
    printf("[%c]", '\a');
    printf("[%c]", '\n');
    printf("[%c]", '\t');
    printf("[%c]", '\0');
    return 0;
}
```

```
[ ][
][      ][]
```


String constants

- String constants are zero or more characters in double quotes.
- An array of chars that has a `NULL` character at the end of the string `'\0'`
  - `char a[]="Hello";` is the same as
  - `char a[]={'H','e','l','l','o','\0'};`


String constants

- String constants are zero or more characters in double quotes.
- An array of chars that has a `NULL` character at the end of the string `'\0'`
  - `char a[]="Hello";` is the same as
  - `char a[]={'H','e','l','l','o','\0'};`

In [42]:

```c
#include <string.h>
#include <stdio.h>

int main(){
    char a[] = "x";
    char b = 'x';
    printf("length of a: %ld character(s)\n", strlen(a)); // returns number of characters
    printf("size of a: %ld byte(s)\n", sizeof(a)); // returns number of bytes
    printf("size of b: %ld byte(s)\n", sizeof(b));
}
```

```
length of a: 1 character(s)
size of a: 2 byte(s)
size of b: 1 byte(s)
```


Enumerations

- In many programs, we'll need variables that have only a small set of meaningful values.
- A variable that stores the suit of a playing card should have only four potential values: "clubs", "diamonds", "hearts", and "spades".


Alternative to enum

- A "suit" variable can be declared as an integer, with a set of codes that represent the possible values of the variable:

```
int s; /* s will store a suit */
...
s = 2; /* 2 represents "hearts" */
```

- Problems with this technique:
  - We can't tell that `s` has only four possible values
  - The significance of `2` isn't apparent


Alternative to enum

- An alternative solution would be to use macros to define a suit "type" and names for the various suits - a step in the right direction:

```
#define SUIT     int
#define CLUBS    0
#define DIAMONDS 1
#define HEARTS   2
#define SPADES   3
```

- An updated version of the previous example:

```
SUIT s;
...
s = HEARTS;
```


Alternative to enum

- Problems with this technique:
  - There's no indication to someone reading the program that the macros represent values of the same "type".
  - If the number of possible values is more than a few, defining a separate macro for each will be tedious.
  - The names `CLUBS`, `DIAMONDS`, `HEARTS` and `SPADES` will be removed by the preprocessor, so they won't be available during debugging.


Enumerations

- C provides a special kind of type designed specifically for variables that have a small number of possible values.
- An *enumerated* type is a type whose values are listed ("enumerated") by the programmer.
- Each value must have a name (an enumeration constant).


Enumerations

- Enumerations are declared like this:

```
enum suit{CLUBS, DIAMONDS, HEARTS, SPADES};
```

- The names of the constants must be different from other identifiers declared in the enclosing scope.
- Enumeration constants are similar to `#define` directive constants, but not equivalent.
- If an enumeration is declared inside a function, its constants won't be visible outside the function.


Enumerations

Example


Enumerations

Example

In [43]:

```c
#include <stdio.h>

enum suit{CLUBS, DIAMONDS, HEARTS, SPADES};

int main(){
        printf("Clubs = int value %d\n",CLUBS);
        printf("Spades = int value %d\n",SPADES);
        enum suit card;
        card = DIAMONDS;
        printf("card is diamond? %d\n",card==DIAMONDS);
}
```

```
Clubs = int value 0
Spades = int value 3
card is diamond? 1
```


Enumerations

- Behind the scenes, C treats enumeration variables and constants as integers.
- By default, the compiler assigns the integers `0`, `1`, `2`, ... to the constants in a particular enumeration.
- In the suit enumeration, `CLUBS`, `DIAMONDS`, `HEARTS` and `SPADES` represent `0`, `1`, `2` and `3`, respectively.


Enumerations as Integers

- The programmer can choose different values for enumeration constants.
- The values of enumeration constants may be arbitrary integers, listed in no particular order:

```
enum dept {RESEARCH = 20, PRODUCTION = 10, SALES = 25};
```

- It is even allowed for two or more enumeration constants to have the same value!


Enumerations as Integers

- When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant
- The first enumeration constant has the value `0` by default
- Example:


Enumerations as Integers

- When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant
- The first enumeration constant has the value `0` by default
- Example:

In [ ]:

```c
#include <stdio.h>

enum EGA_colors {BLACK, LT_GRAY = 7, DK_GRAY, WHITE = 15};

int main(){
        printf("Black = %d\n",BLACK);
        printf("Light Gray = %d\n",LT_GRAY);
        printf("Dark Gray = %d\n",DK_GRAY);
        printf("White = %d\n",WHITE);
}
```


Functions in C - declaration

- Functions encapsulate code in a convenient way.
- Analogous to methods in an O-O language.
- Functions can be defined anywhere in a program file, if the declaration precedes use of the function.


Functions in C - declaration

- Functions encapsulate code in a convenient way.
- Analogous to methods in an O-O language.
- Functions can be defined anywhere in a program file, if the declaration precedes use of the function.

In [39]:

```c
#include <stdio.h>
int power( int base, int n ) {

     int p;
     for ( p = 1; n > 0; n-- )
         p = p * base;
     return p;
}

int main() {
    int result = power(2,3);
    printf("The result is %d", result);
    return 0;
}
```

```
The result is 8
```


Functions in C - declaration

- Functions can be *declared* before they are defined, as a function declaration:

```
return-type function-name ( parameters );
```

- e.g. to calculate `base` raised to the power `n`

```
int power( int base, int n );
```

- the input parameters (`n` and `base`) do not need to be named when the function is declared! They need names when the function is defined.
- Often we put these in a header file (`.h`)


Functions in C: call by value

- Function parameters in C are passed using a call-by-value semantic.

```
result = power(x, y);
```

- Here, when `x` and `y` are passed through to `power()`, the values of `x` & `y` are copied to the `base` and `n` variables in the function.
- A function cannot affect the value of its arguments!


Functions in C: call by value

`swap(x,y)` example


Functions in C: call by value

`swap(x,y)` example

In [40]:

```c
#include <stdio.h>

void swap(int a, int b);

int main(){
    int x = 8;
    int y = 44;
    printf("pre- swap: x = %d  y = %d\n", x, y);
    swap(x,y);
    printf("post swap: x = %d  y = %d\n", x, y);
    return 0;
}

void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

```
pre- swap: x = 8  y = 44
post swap: x = 8  y = 44
```


Functions in C - declaration

- Functions encapsulate code in a convenient way.
- Analogous to methods in an O-O language.
- Functions can be defined anywhere in a program file, if the declaration precedes use of the function.


Functions in C - declaration

- Functions encapsulate code in a convenient way.
- Analogous to methods in an O-O language.
- Functions can be defined anywhere in a program file, if the declaration precedes use of the function.

In [39]:

```c
#include <stdio.h>
int power( int base, int n ) {

     int p;
     for ( p = 1; n > 0; n-- )
         p = p * base;
     return p;
}

int main() {
    int result = power(2,3);
    printf("The result is %d", result);
    return 0;
}
```

```
The result is 8
```


Functions in C - declaration

- Functions can be *declared* before they are defined, as a function declaration:

```
return-type function-name ( parameters );
```

- e.g. to calculate `base` raised to the power `n`

```
int power( int base, int n );
```

- the input parameters (`n` and `base`) do not need to be named when the function is declared! They need names when the function is defined.
- Often we put these in a header file (`.h`)


Functions in C: call by value

- Function parameters in C are passed using a call-by-value semantic.

```
result = power(x, y);
```

- Here, when `x` and `y` are passed through to `power()`, the values of `x` & `y` are copied to the `base` and `n` variables in the function.
- A function cannot affect the value of its arguments!


Functions in C: call by value

`swap(x,y)` example


Functions in C: call by value

`swap(x,y)` example

In [40]:

```c
#include <stdio.h>

void swap(int a, int b);

int main(){
    int x = 8;
    int y = 44;
    printf("pre- swap: x = %d  y = %d\n", x, y);
    swap(x,y);
    printf("post swap: x = %d  y = %d\n", x, y);
    return 0;
}

void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

```
pre- swap: x = 8  y = 44
post swap: x = 8  y = 44
```


Functions in C: Organisation

- We can use Header files .h and Source files .c to organise our code


Functions in C: Organisation

- We can use Header files .h and Source files .c to organise our code

swap.h


Functions in C: Organisation

- We can use Header files .h and Source files .c to organise our code

swap.h

In [40]:
```
1 void swap(int a, int b);
```

```
pre- swap: x = 8  y = 44
post swap: x = 8  y = 44
```


Functions in C: Organisation

- We can use Header files .h and Source files .c to organise our code

swap.h

In [40]:
```
1 void swap(int a, int b);
```

```
pre- swap: x = 8  y = 44
post swap: x = 8  y = 44
```

swap.c


Functions in C: Organisation

- We can use Header files .h and Source files .c to organise our code

swap.h

In [40]:
```
1 void swap(int a, int b);
```

```
pre- swap: x = 8  y = 44
post swap: x = 8  y = 44
```

swap.c

In [40]:
```
1 #include "swap.h"
2
3 void swap(int a, int b) {
4     int temp = a;
5     a = b;
6     b = temp;
7 }
```

```
pre- swap: x = 8  y = 44
post swap: x = 8  y = 44
```

Functions in C: Organisation

- We can use Header files .h and Source files .c to organise our code

swap.h

In [40]:
```
1 void swap(int a, int b);
```

```
pre- swap: x = 8  y = 44
post swap: x = 8  y = 44
```

swap.c

In [40]:
```
1 #include "swap.h"
2
3 void swap(int a, int b) {
4     int temp = a;
5     a = b;
6     b = temp;
7 }
```

```
pre- swap: x = 8  y = 44
post swap: x = 8  y = 44
```

Functions in C: Organisation

- We can use Header files .h and Source files .c to organise our code

swap.h

In [40]:
```c
void swap(int a, int b);
```

```
pre- swap: x = 8  y = 44
post swap: x = 8  y = 44
```

swap.c

In [40]:
```c
#include "swap.h"

void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

```
pre- swap: x = 8  y = 44
post swap: x = 8  y = 44
```


Summary

- Switch statements
- Increment and Decrement operators
- Data types
- Enums
- Intruduction to functions