Programming Paradigms

CT331 Week 2 Lecture 2

Finlay Smith

finlay.smith@universityofgalway.ie

Imperative and Procedural Programming

Imperative Programming

- Eg. Assembly, (also Fortran and Basic, sometimes)
    - List of instructions
    - GOTO statements
    - Little or no structure

Telling the computer to perform a set of actions, one after the other.

Most programming languages have imperative aspects.

Procedural Programming

- Eg. C, Pascal, Ada (also Fortran and Basic, sometimes)
    - Code is structured
    - Uses functions, or procedures *
    - Encourages code re-use
    - Encourages encapsulation and composition.

Splits actions into procedures or tasks.

Procedures can be made up of other procedures. Composition (Recursion…)

*Procedural functions are distinct from “functional programming”

Structured Programming

- Eg. Basically everything but Assembly
  - Code is structured
  - While, For, if, else, switch, class, function, etc.
  - Less emphasis on GOTO statements.

- Creating a structure to manage instructions.
- Allows more complex programs to be built.
- Easier to understand.
- Avoids GOTO bugs and spaghetti code.

The C programming Language

The C programming language

- Procedural, Imperative, Structured, “systems language”
- Came into being in 1969-1973 in parallel with the development of the unix OS
- BCPL -> B -> C -> …
- ANSI standard (since 1980s)
- Now one of the most popular (and powerful) languages in use today.

Thompson was faced with a hardware environment cramped and spartan even for the time: the DEC PDP-7 on which he started in 1968 was a machine with 8K 18-bit words of memory and no software useful to him. While wanting to use a higher-level language, he wrote the original Unix system in PDP-7 assembler. At the start, he did not even program on the PDP-7 itself, but instead used a set of macros for the GEMAP assembler on a GE-635 machine. A postprocessor generated a paper tape readable by the PDP-7.

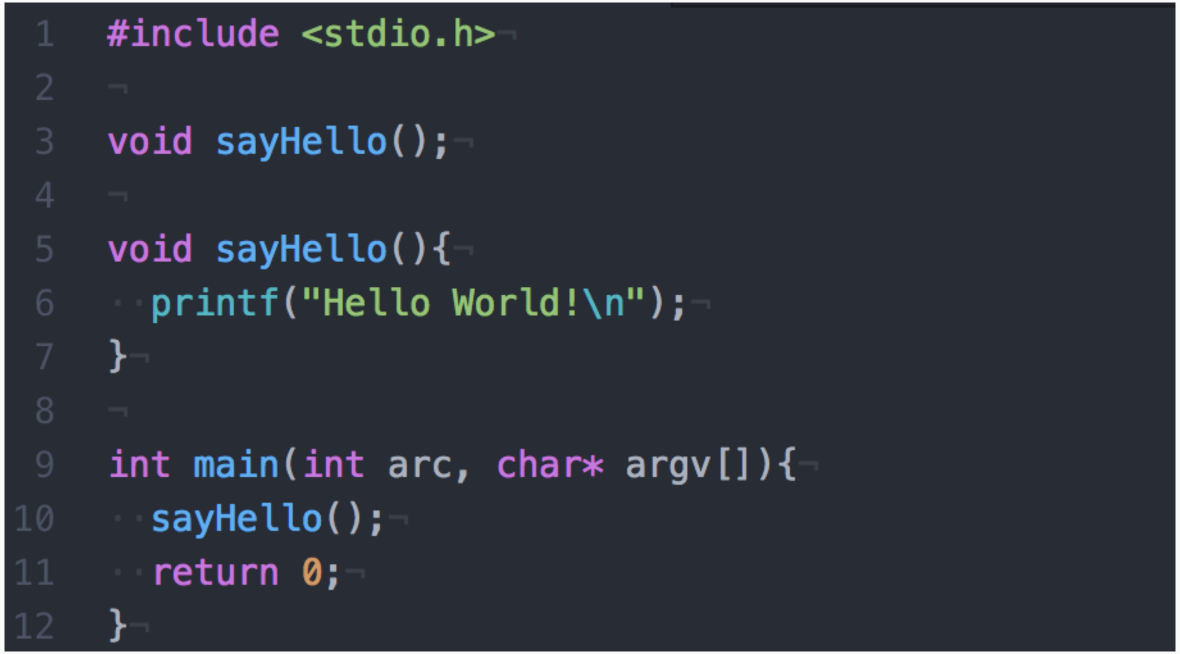
These tapes were carried from the GE machine to the PDP-7 for testing until a primitive Unix kernel, an editor, an assembler, a simple shell (command interpreter), and a few utilities (like the Unix *rm, cat, cp* commands) were completed. After this point, the operating system was self-supporting: programs could be written and tested without resort to paper tape, and development continued on the PDP-7 itself.

-Denis M. Richie, The Development of the C Programming Language, Bell Labs.
https://www.bell-labs.com/usr/dmr/www/chist.html

BCPL, B, and C all fit firmly in the traditional procedural family typified by Fortran and Algol 60. They are particularly oriented towards system programming, are small and compactly described, and are amenable to translation by simple compilers. They are `close to the machine' in that the abstractions they introduce are readily grounded in the concrete data types and operations supplied by conventional computers, and they rely on library routines for input-output and other interactions with an operating system.

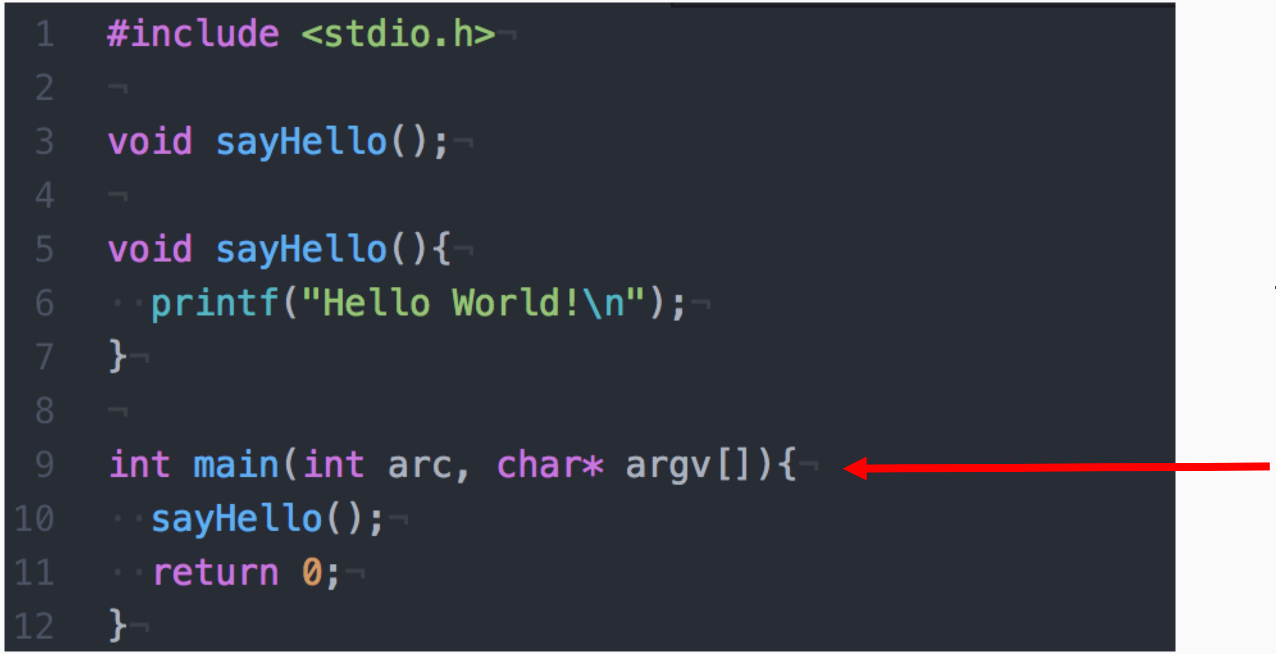
-Denis M. Richie, The Development of the C Programming Language, Bell Labs.
https://www.bell-labs.com/usr/dmr/www/chist.html

An Example C Program: helloWorld.c



```c
#include <stdio.h>

void sayHello();

void sayHello(){
  printf("Hello World!\n");
}

int main(int arc, char* argv[]){
  sayHello();
  return 0;
}
```

An Example C Program: helloWorld.c



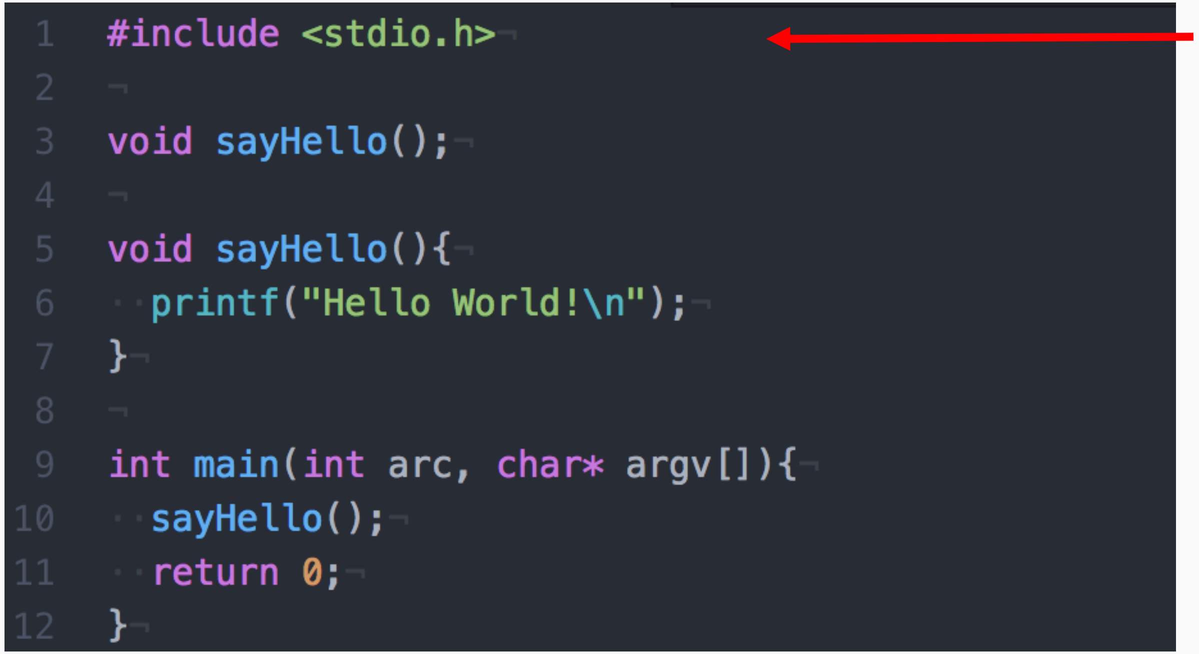
```
#include <stdio.h>

void sayHello();

void sayHello(){
  printf("Hello World!\n");
}

int main(int arc, char* argv[]){
  sayHello();
  return 0;
}
```

Main function:
Entry point to the program

Returns int.

Two arguments:
arg: number of command line arguments.
argv: the arguments.

An Example C Program: helloWorld.c



```
#include <stdio.h>

void sayHello();

void sayHello(){
  printf("Hello World!\n");
}

int main(int arc, char* argv[]){
  sayHello();
  return 0;
}
```

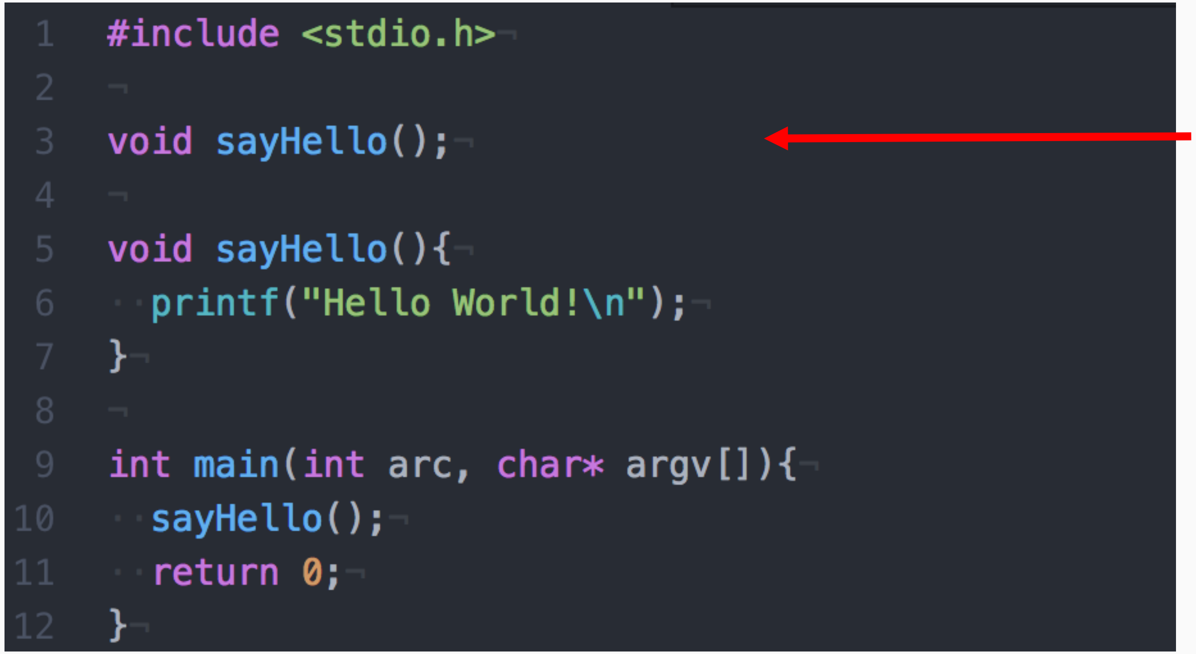
Header inclusion

Functionality defined in stdio.h is added into helloWorld.c by the compiler.

(specifically the Linker step of compiler)

Stdio.h = “Standard Input/Output”

An Example C Program: helloWorld.c



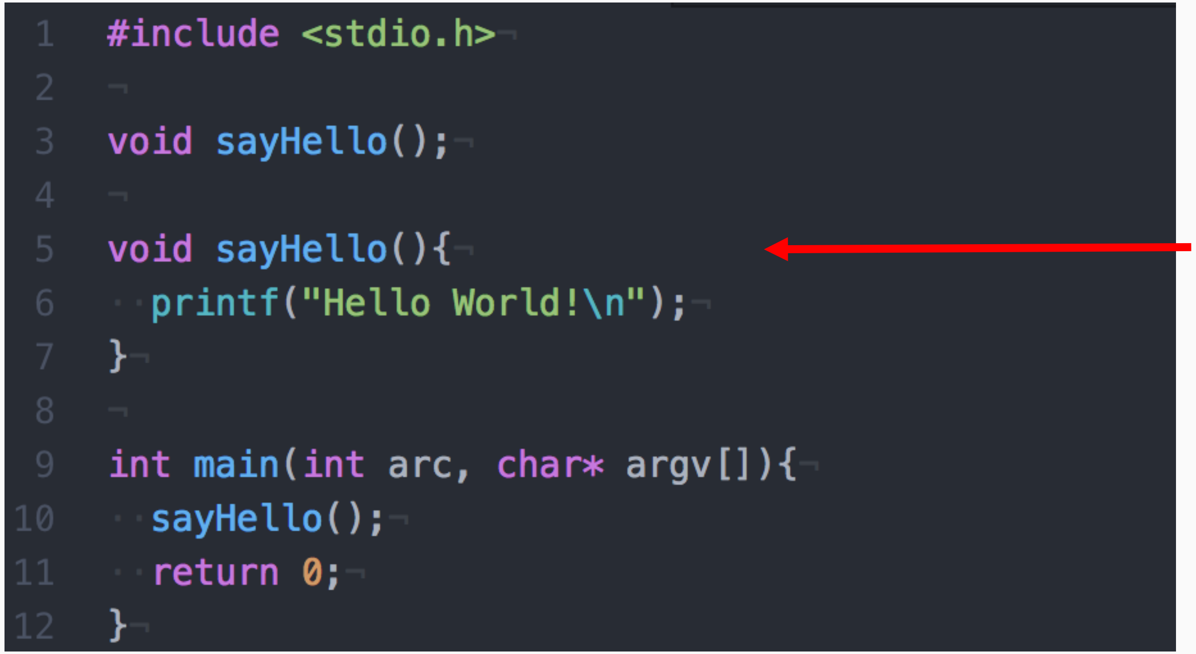
```
#include <stdio.h>

void sayHello();

void sayHello(){
  printf("Hello World!\n");
}

int main(int arc, char* argv[]){
  sayHello();
  return 0;
}
```

Function Prototype.

Tells compiler that the function exists before it has been implemented.

Allows the compiler to handle recursion, or functions calling each other.

An Example C Program: helloWorld.c



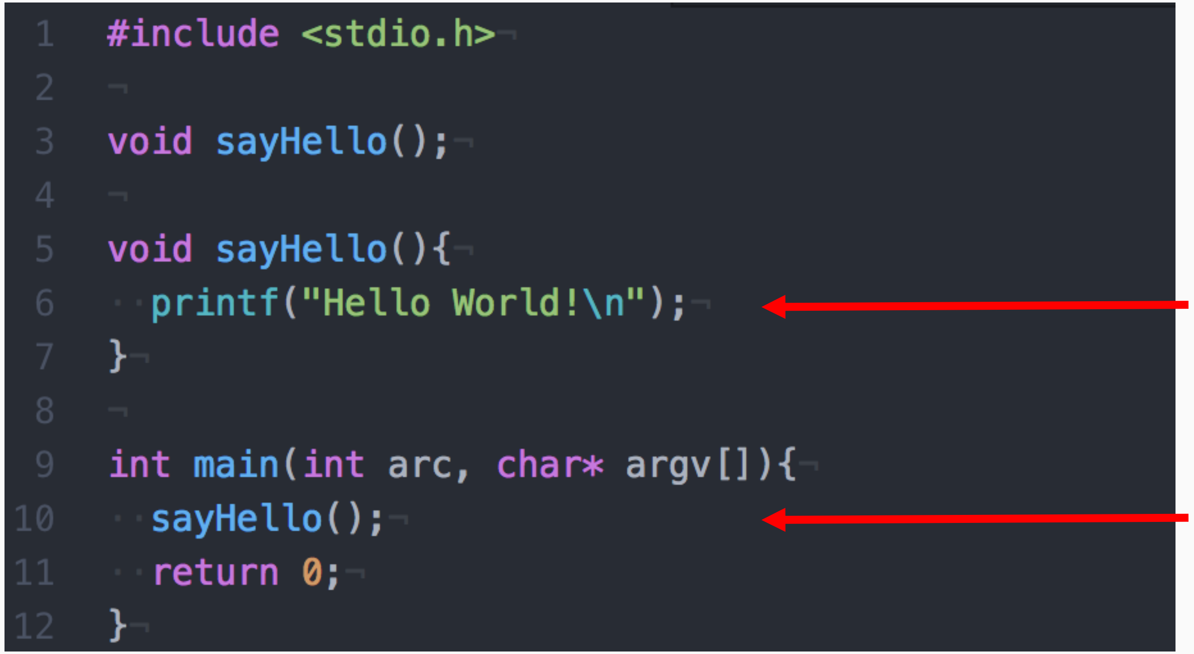
```
#include <stdio.h>

void sayHello();

void sayHello(){
  printf("Hello World!\n");
}

int main(int arc, char* argv[]){
  sayHello();
  return 0;
}
```

Function Definition.

Implements the function.

<u>Note:</u>
Data type
Arguments…
return...

An Example C Program: helloWorld.c



```c
#include <stdio.h>

void sayHello();

void sayHello(){
  printf("Hello World!\n");
}

int main(int arc, char* argv[]){
  sayHello();
  return 0;
}
```

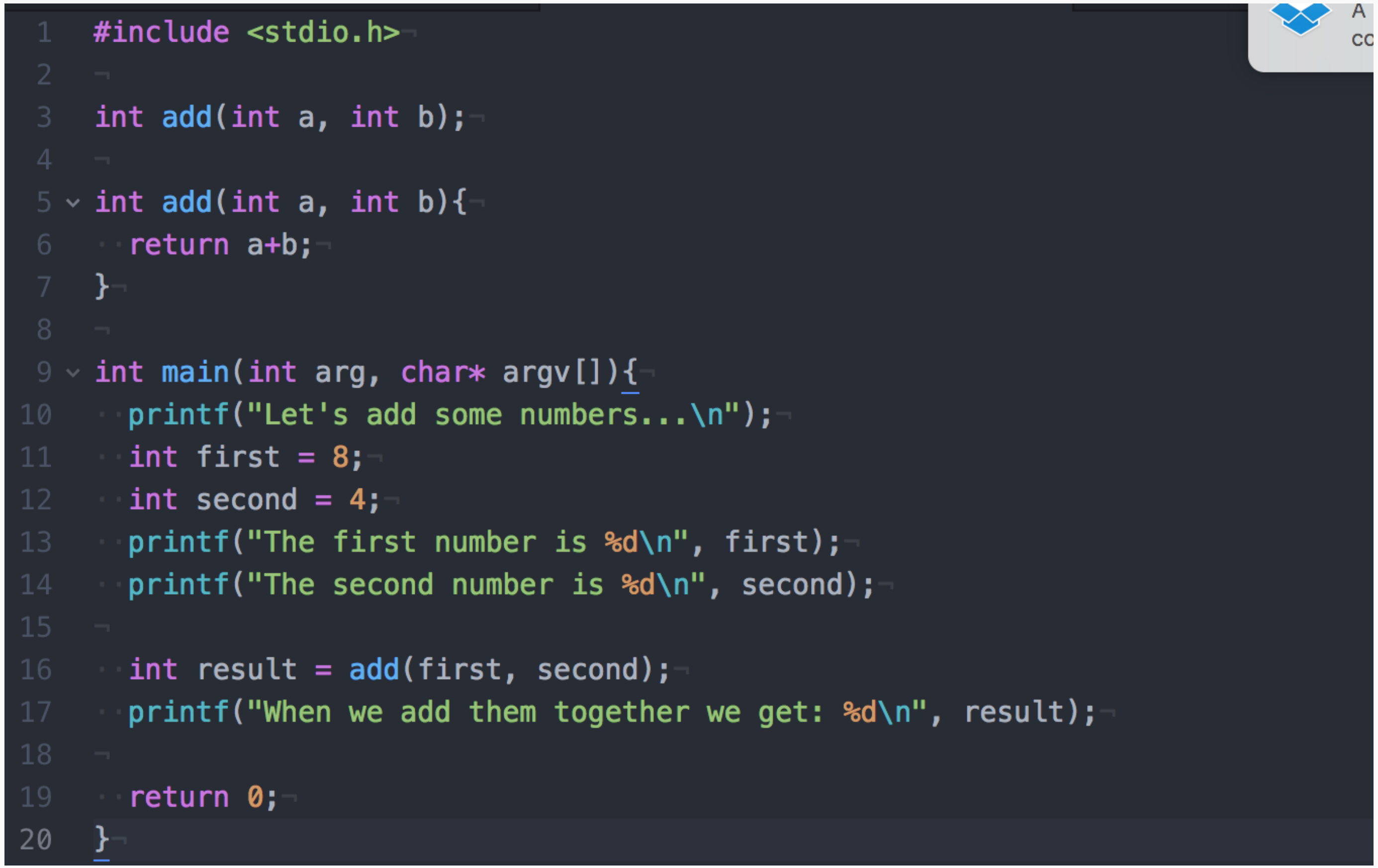
Calling a function.

Printf takes a char* argument.

sayHello takes no argument.

Nothing being returned.

An Example C Program: addNumbers.c



```c
#include <stdio.h>

int add(int a, int b);

int add(int a, int b){
  return a+b;
}

int main(int arg, char* argv[]){
  printf("Let's add some numbers...\n");
  int first = 8;
  int second = 4;
  printf("The first number is %d\n", first);
  printf("The second number is %d\n", second);

  int result = add(first, second);
  printf("When we add them together we get: %d\n", result);

  return 0;
}
```

“add” is a function that returns an int.

Int is stored in the “result” variable.

Must have same data type.

<u>Note:</u>

Printf with multiple parameters…

%d for ints – strictly decimal ints (%i is any int including oct and hex)

Pointers

```
int* p; // variable p is pointer to integer type

int i; // integer value
```

Pointers

You turn a <u>pointer</u> into a <u>value</u> with *

```
int i2 = *p;
```

// integer i2 is assigned with integer value that pointer p is pointing to

Pointers

You turn a value into a pointer with &:

```
int* p2 = &i;
```

// pointer p2 will point to the address of integer i

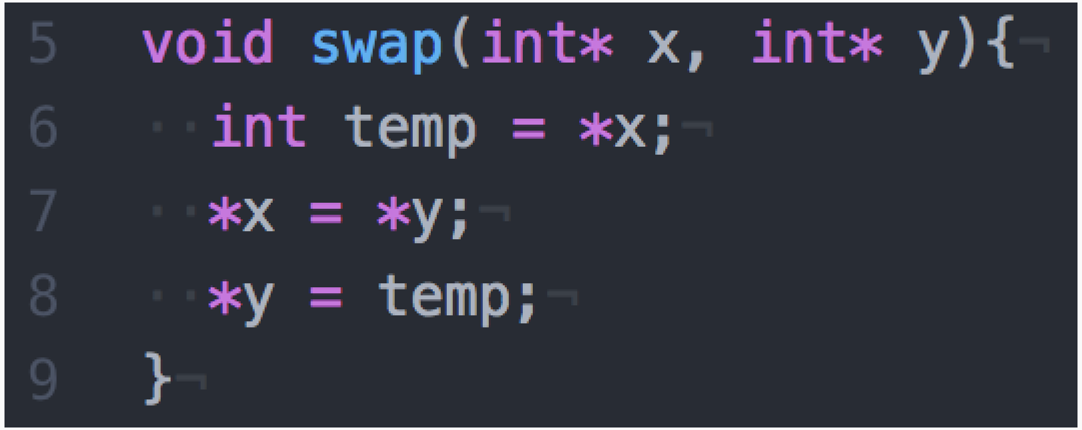
Using Pointers (side effects!)

```
int a = 8;
int b = 4;
swap(&a, &b);
```

(should make a=4 and b=8)

A function effectively breaking the convention that args are not changed in a function is a side effect – done by passing addresses

Using Pointers (side effects!)



```c
void swap(int* x, int* y){
  int temp = *x;
  *x = *y;
  *y = temp;
}
```

Arrays and Pointers

`int intArr[5];  // an integer array of size 5`

`intArr` is a pointer to 1st element of array

- same as `&intArr[0]`

`intArr[2]=3;`

- Same as `*(intArr+2) = 3;`

`(intArr + 2)` is of type (int*)

- while `intArr[2]` is of type int.
- Latter case - the pointer is dereferenced

`(intArr + 2)` is same as `(&(intArr[2]))`

```
Note: the + operator here is not simple addition – it
moves the pointer by the size of the type
```

Generic swap function?

What about a swap function that works on any data type?

```
???????

void swap(void* x, void* y){
  void temp = *x;
  *x = *y;
  *y = temp;
}

???????
```

Generic swap function?

What about a swap function that works on any data type?

```
???????

void swap(void* x, void* y){
   void temp = *x;
   *x = *y;
   *y = temp;
}

???????
```

We don’t know what size data *x point to…

So void temp can’t work.

It is impossible to have a variable of type void for this reason.

But we can have a pointer of type void*

void*

- Is a specific pointer type - void *
- Points to *some* location in memory
- No specific type
- Therefore: No specific size

sizeof()

`sizeof( type )`

Returns the size, in bytes, of the object representation of `type`

`(Built in to C language)`

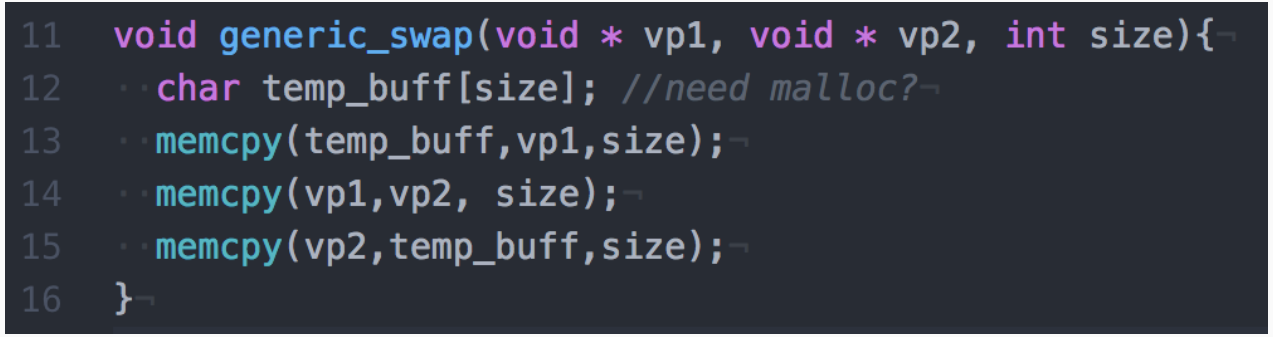
memcpy()

`void * memcpy (void * to, const void *from, size_t size)`

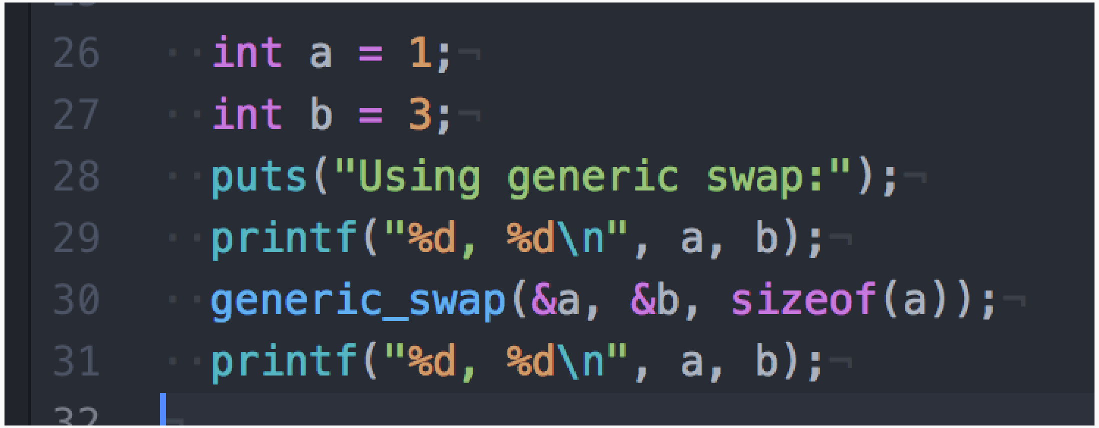
The memcpy function copies `size` bytes from the object beginning at `from` into the object beginning at `to`. The value returned by memcpy is the value of `to`.

(Not built in. Defined in string.h - `#include <string.h>`)

Generic Swap function



```c
void generic_swap(void * vp1, void * vp2, int size){
  char temp_buff[size]; //need malloc?
  memcpy(temp_buff,vp1,size);
  memcpy(vp1,vp2, size);
  memcpy(vp2,temp_buff,size);
}
```



```c
  int a = 1;
  int b = 3;
  puts("Using generic swap:");
  printf("%d, %d\n", a, b);
  generic_swap(&a, &b, sizeof(a));
  printf("%d, %d\n", a, b);
```