Procedures, Parameters, Values and Variables

Steven R. Bagley

Recap

- A Program is a sequence of statements (instructions)
- Statements executed one-by-one in order
- Unless it is changed by the programmer e.g. by calling a function

Function Anatomy

```
return-type function-name(parameter declarations)
{
    declarations
    statements
}
```

```
/* Defines a function called PrintHello() */
void PrintHello()
{
    printf(“Hello World\n”);
}

int main(int argc, char *argv[])
{
    PrintHello(); /* Call PrintHello function */
}
```

Note the semicolons -- they are significant
void means doesn’t return anything

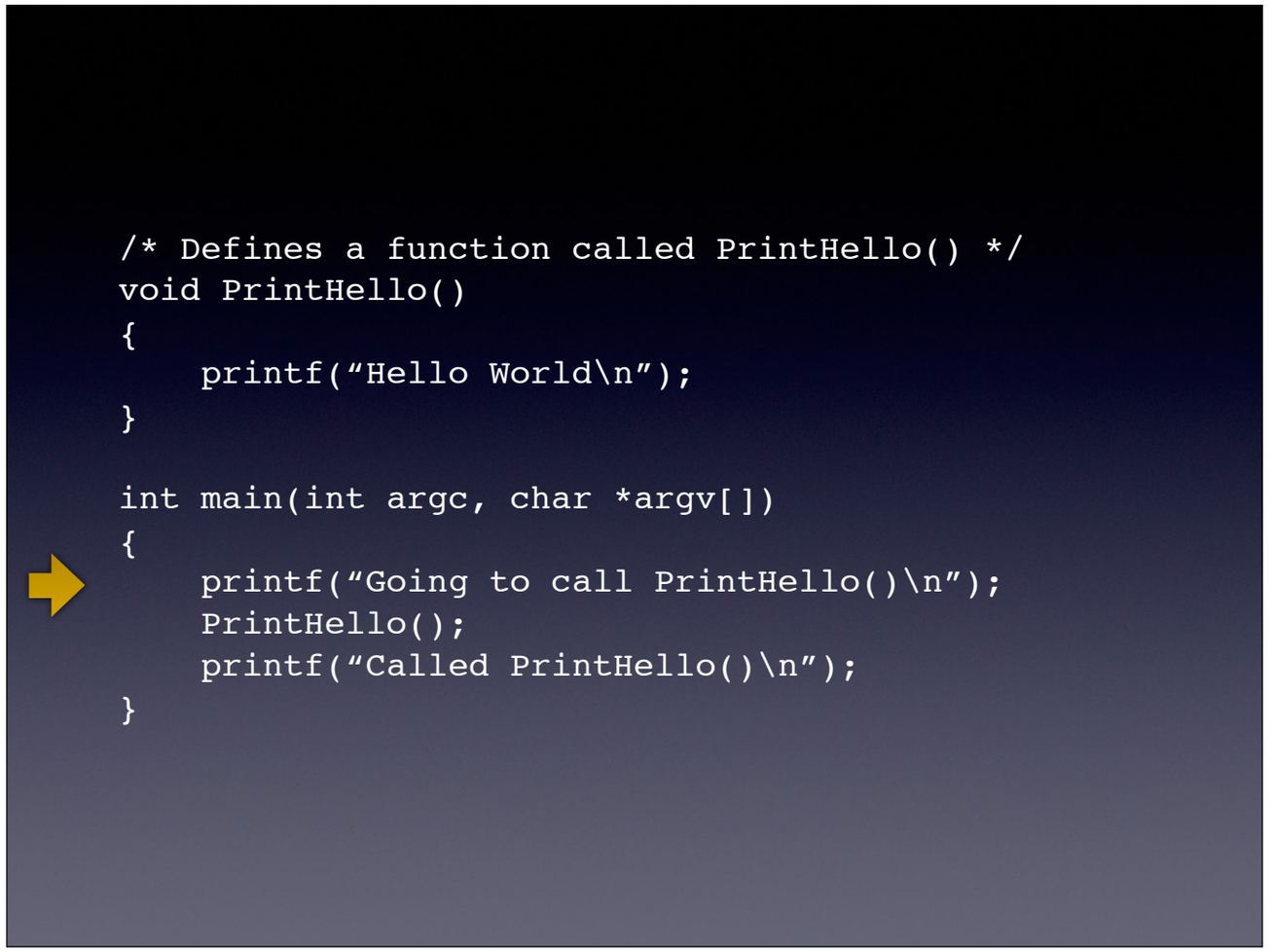
```
/* Declare function called PrintHello() exists */
void PrintHello();

int main(int argc, char *argv[])
{
    PrintHello(); /* Call PrintHello function */
}

/* Define PrintHello() function */
void PrintHello()
{
    printf("Hello World\n");
}
```

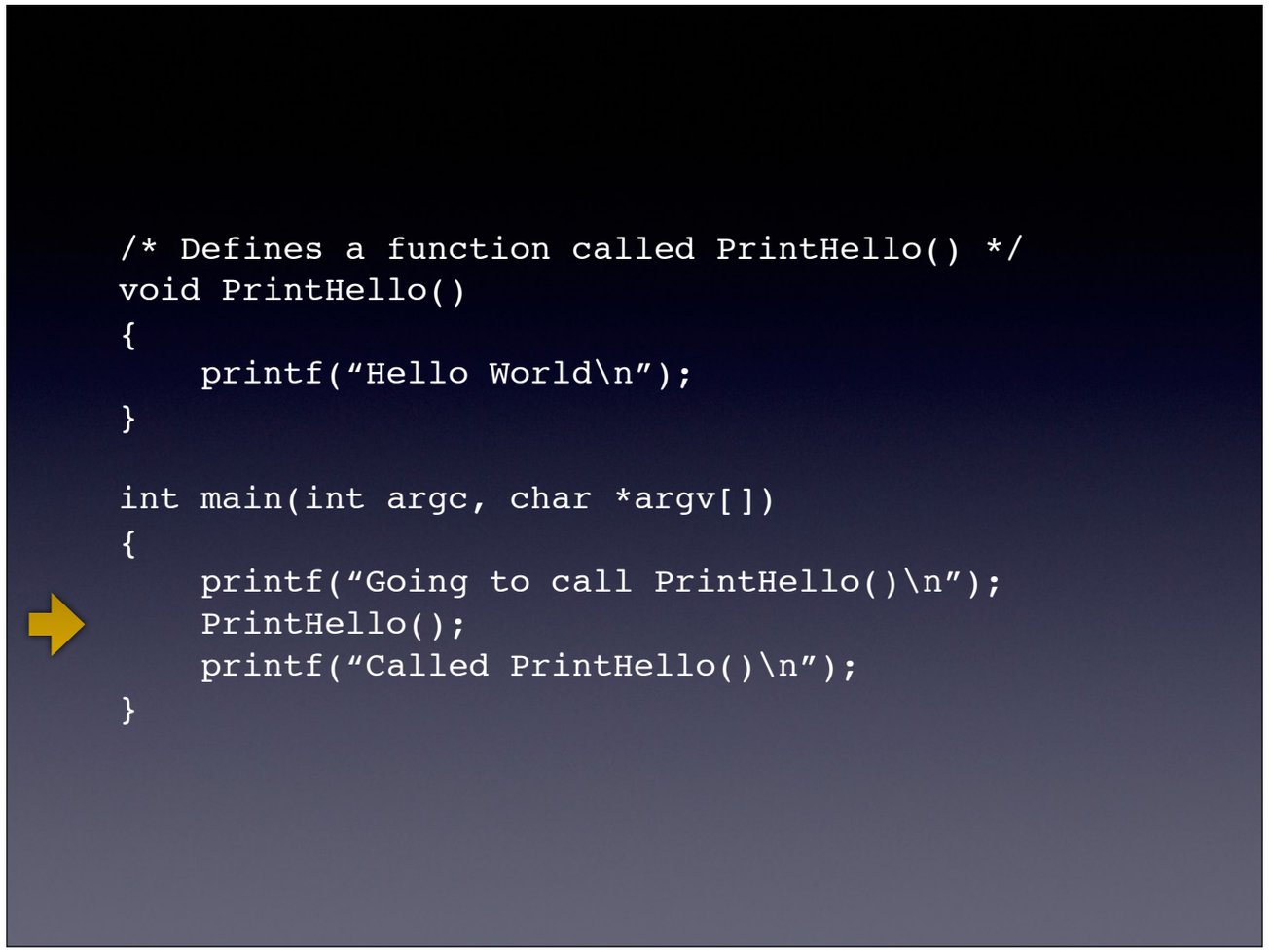
These must match

Note the semicolons -- they are significant
void means doesn't return anything
IF they don't match, you'll get compile errors

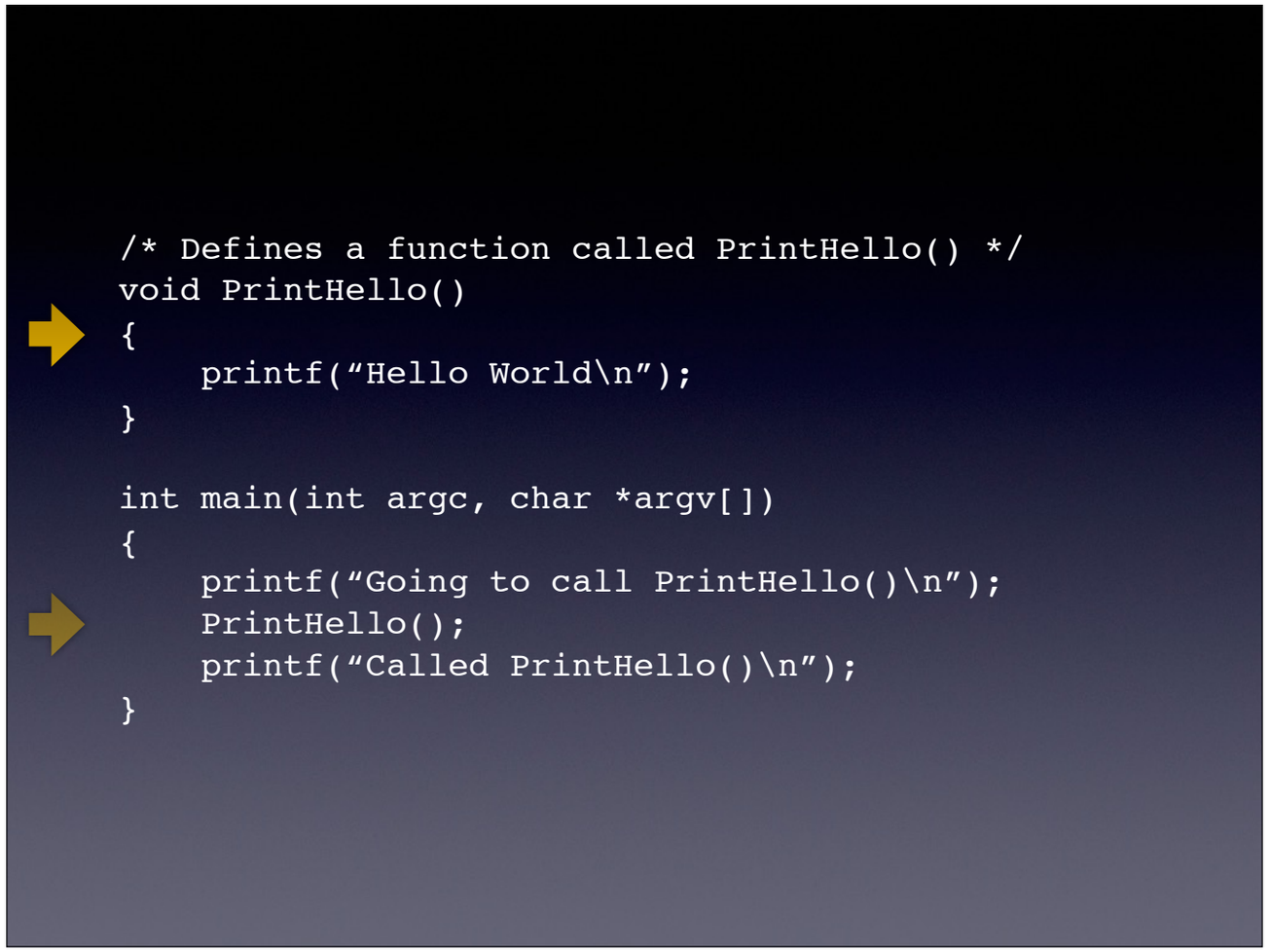
Calling Functions

- When we call a function:
  - Stops executing the current function
  - Starts executing the code in the function called
  - When finished, it continues executing where it left off in the original function

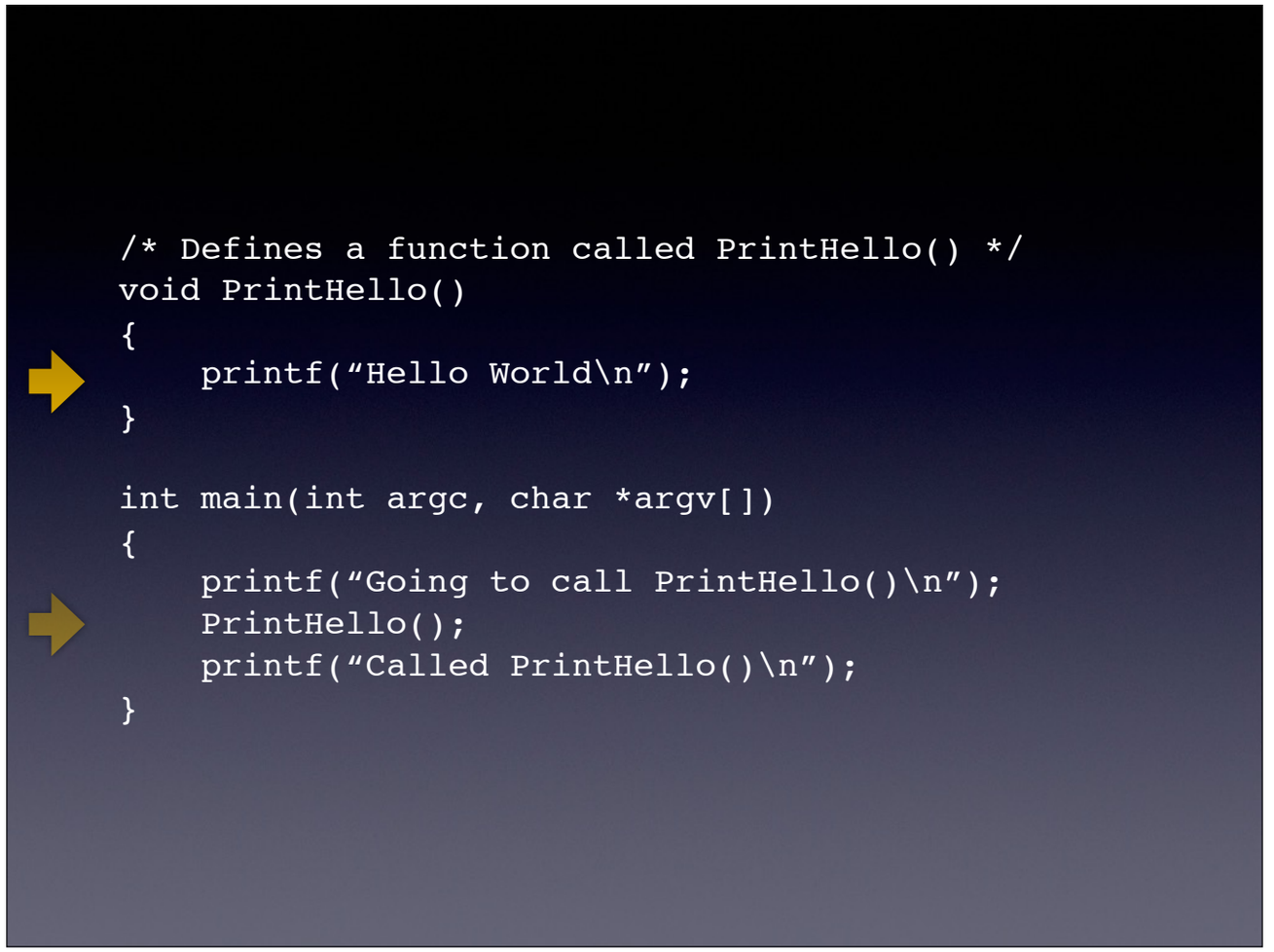

/* Defines a function called PrintHello() */
void PrintHello()
{
printf(“Hello World\n”);
}
int main(int argc, char *argv[])
{
printf(“Going to call PrintHello()\n”);
PrintHello();
printf(“Called PrintHello()\n”);
}

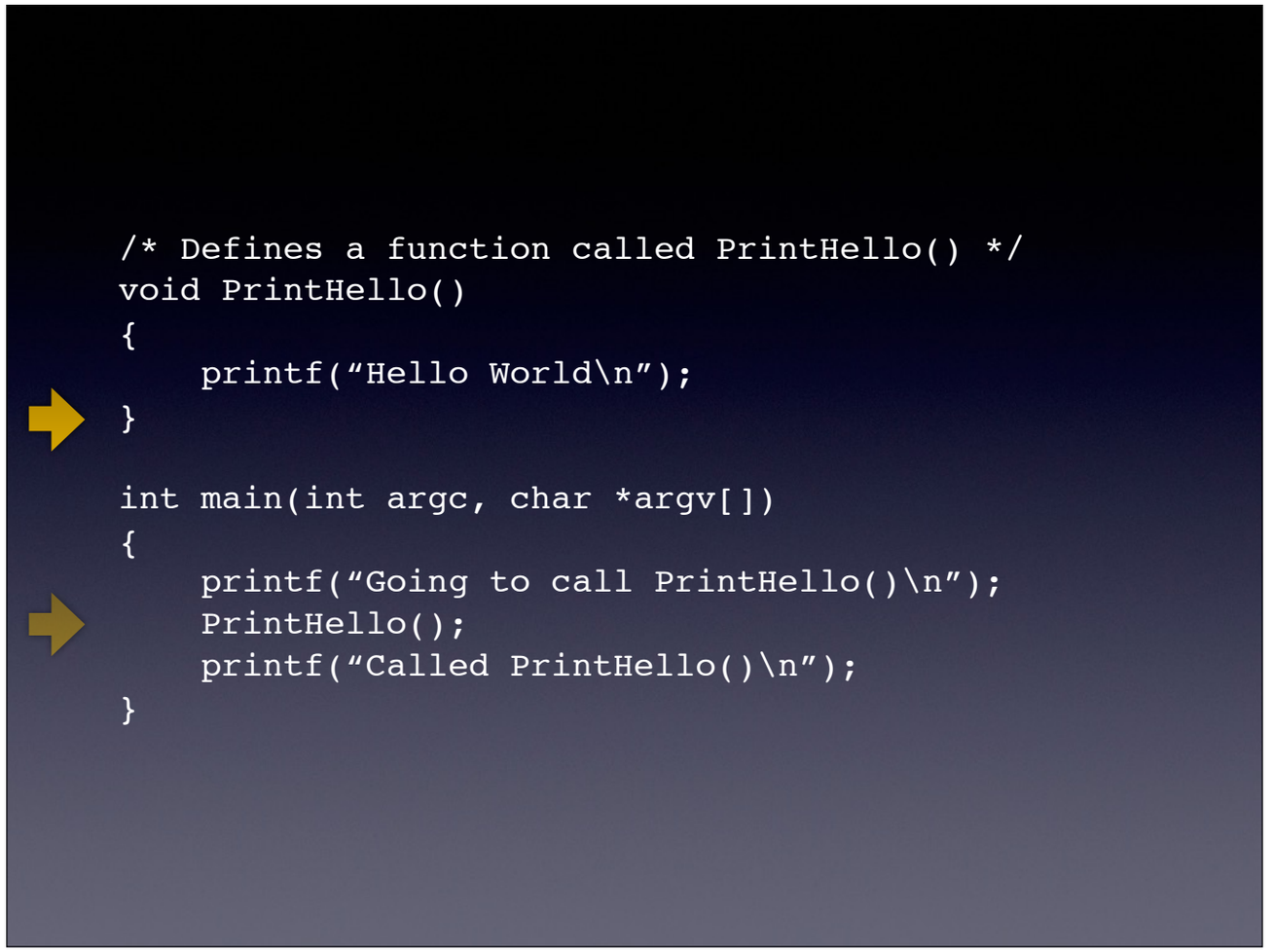


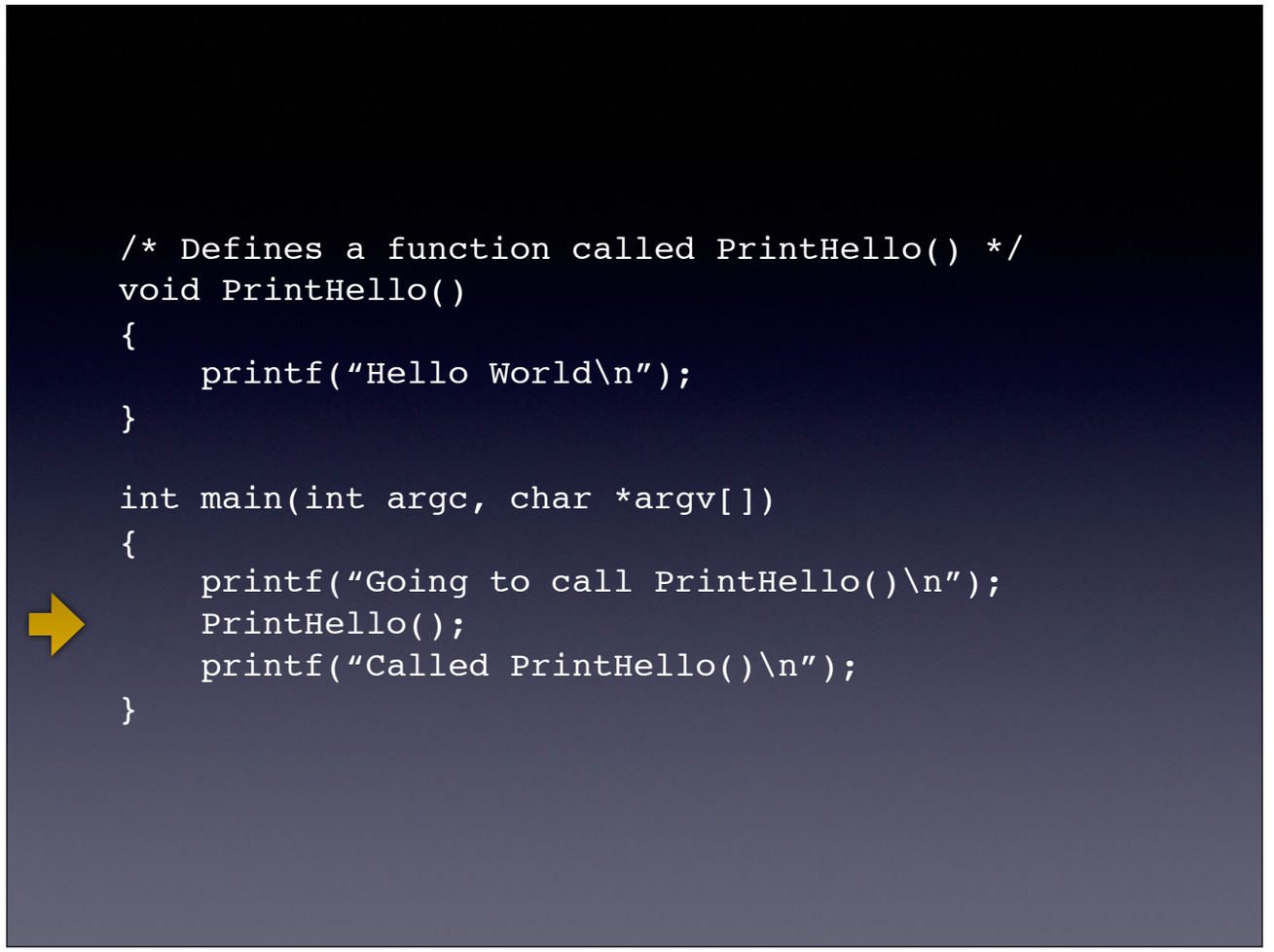
```
/* Defines a function called PrintHello() */
void PrintHello()
{
    printf("Hello World\n");
}

int main(int argc, char *argv[])
{
    printf("Going to call PrintHello()\n");
    PrintHello();
    printf("Called PrintHello()\n");
}
```



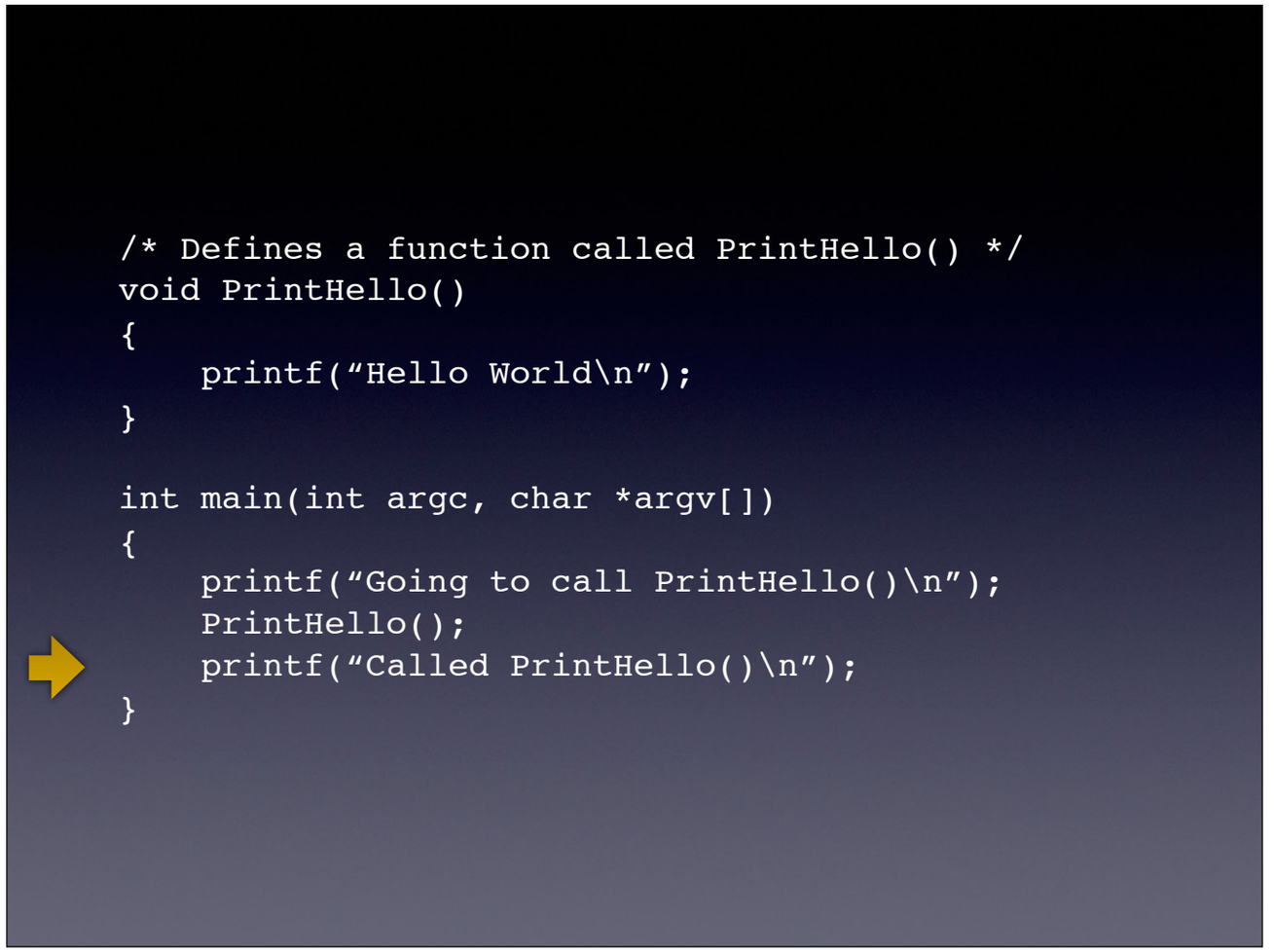
```
/* Defines a function called PrintHello() */
void PrintHello()
{
    printf("Hello World\n");
}

int main(int argc, char *argv[])
{
    printf("Going to call PrintHello()\n");
    PrintHello();
    printf("Called PrintHello()\n");
}
```


/* Defines a function called PrintHello() */
void PrintHello()
{
printf(“Hello World\n”);
}
int main(int argc, char *argv[])
{
printf(“Going to call PrintHello()\n”);
PrintHello();
printf(“Called PrintHello()\n”);
}


/* Defines a function called PrintHello() */
void PrintHello()
{
printf(“Hello World\n”);
}
int main(int argc, char *argv[])
{
printf(“Going to call PrintHello()\n”);
PrintHello();
printf(“Called PrintHello()\n”);
}



```
/* Defines a function called PrintHello() */
void PrintHello()
{
    printf("Hello World\n");
}

int main(int argc, char *argv[])
{
    printf("Going to call PrintHello()\n");
    PrintHello();
    printf("Called PrintHello()\n");
}
```


/* Defines a function called PrintHello() */
void PrintHello()
{
printf(“Hello World\n”);
}
int main(int argc, char *argv[])
{
printf(“Going to call PrintHello()\n”);
PrintHello();
printf(“Called PrintHello()\n”);
}

Program Flow

- This is important
- Program usually flows from one statement to the next
- But it doesn't have to
- The power of programming comes from controlling the order of execution
- Which we'll see more of later…

We'll see more of this later in the course

`#include` files

- Can end up declaring a lot of procedures
- Some of these can be in other files…
- Rather than typing them all in, we `#include` them from elsewhere
- Copies content of file (usually a header file — ending with `.h`) into the program, e.g. `#include <stdio.h>`

stdio.h is a standard library header file

`#include` files

- Searches for the file in various locations, depending on format…
- Search in system directories
  `#include <stdio.h>`
- Search in current directory (and system directory)
  `#include “stdio.h”`

if the file is in <> system, if in quotes current + system

Data

- Programs = Algorithms + Data
- Algorithms manipulate the data to perform their task
- In C, data is stored in variables (or in things accessed via variables)
- Algorithms in C are implemented by manipulating the values stored in variables

Algorithm == the code we write. Several standard ones that are often used (e.g. several to sort things)

Variables

- Variables are a place to store some data
- Given a name that we can use to access it
- Usually only a very small amount of data (a number, or a character)
- Value can change (vary) as program runs

In C anyway, some other languages have single assignment variables
Name is for the place where a value is stored, not the value

Variable Lifetime

- Some values are stored for a long time
  - Player's position in a game
  - Used and accessed in lots of places
- Other values are only needed for a short period
  - To calculate temporary values

Average Algorithm

```
sum = 0;
count = 0;

while(moreValues())
{
  sum = sum + nextValue();
  count = count + 1;
}

average = sum / count;
```

THIS IS NOT VALID C, this is psuedo code
Only need sum and count while calculating the average -- after that it can be thrown away

Variable Type

- In C, variables have *type*
- This type defines what the variable can store
- C provides some basic types
- Possible to define our own complex types
- And specify where they are stored

C Types

Type	Stores	Size
`char`	a single byte, holds one ASCII character	8 bits
`int`	An Integer	32 bits
`short`	An Integer	16 bits
`long`	An Integer	64 bits
`long long`	An Integer	64 bits
`float`	single-precision floating point (e.g. 3.141)	32 bits
`double`	double-precision floating point	64 bits

Note size of highlighted types isn't fixed

Size and Signedness

- Integers in C can be either signed or unsigned — prefix with `unsigned` to specify
- Compiler/OS defines size of integer types
- Not defined in the standard
- `short` <= `int` <= `long`
- Values in table as found on 64-bit Linux

size is number of bits they can store, which also means the size of the largest number they can store
Windows has longs as 32bit

Declaring Variables

- Variables must be declared in the declaration section before use
- Declared by putting the type (e.g. `int`)
- Followed by the name (e.g. `fred`) but use something sensible
- Names must begin with a letter, followed by any number of letters or digits

The underscore character _ is also classed as a letter

Function Anatomy

```
return-type function-name(parameter declarations)
{
    declarations
    statements
}
```

Variables need to be declared in the declaration section

Declaring Variables

- Variables must be declared in the declaration section before use
- Declared by putting the type (e.g. `int`)
- Followed by the name (e.g. `fred`) but use something sensible
- Names must begin with a letter, followed by any number of letters or digits

The underscore character _ is also classed as a letter

Reserved Keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Can’t use any of these as names for a variable...

Naming Conventions

- C convention is to use lowercase variable names
- Use the underscore or camel case to improve readability:
  `characterreadfromkeyboard`
  `character_read_from_keyboard`
  `characterReadFromKeyboard`

Naming Conventions

- Use meaningful names
- Related to what the variable stores
- Shorter names for local variables or loop indices (`i` is often used for loops)

```
double celsius; /* Declares a variable called celsius as a double */

int count; /* Declares a variable called count as an int */

long long bigNumber; /* Declares bigNumber as a long long */

short smallNumber; /* Declares smallNumber as a short */

char c; /* Declares a variable called c as a char */
```

Assigning values

- The values stored in a variable is set using the assignment operator =
- Give the variable name, followed by the assignment operator, followed by the value
- Can be combined with the declaration to set an initial value

```
double celsius; /* Declares a variable called celsius as a double */
int count; /* Declares a variable called count as an int */

celsius = 32.5; /* sets the value of celsius to 32.5 */
count = 3; /* set count to 3 */

int newVal = 4; /* Declares a new variable called newVal with the initial
value 4 */

celsius = 16.5; /* change the value in celsius to 16.5 */
```

If a new value is set to a variable then the old value is overwritten
No way to get it back, unless you store it somewhere else.

Values

- A value assigned to a variable can be a number
- But it can also be expressed in terms of another variable or the return value from a function (providing the types match)

```
/* Declares a variable called celsius as a double with the value 32.5 */
double celsius = 32.5;
double temperature;

/* set the value of temperature to be the same as the value of celsius */
temperature = celsius;

celsius = 16.5; /* change the value in celsius to 16.5 */
```

What's the value of `temperature`?

Value copied between variables -- they are two separate entities...

```
/* Declares a variable called celsius as a double with the value 32.5 */
double celsius = 32.5;
double temperature;

/* set the value of temperature to be the same as the value of celsius */
temperature = celsius;

celsius = 16.5; /* change the value in celsius to 16.5 */
```

What's the value of `temperature`?

`32.5`

Value copied between variables -- they are two separate entities...

Values

- In C, values have type too
- `3` is an integer, `3.0` is a double
- This can cause some interesting side-effects

```
double f = 3 / 2;
```

- What is the value of the variable `f`?

f is 1.0000 (the division is between two integers, so divided as integers)

Values

- In C, values have type too
- `3` is an integer, `3.0` is a double
- This can cause some interesting side-effects

```
double f = 3.0 / 2;
```

- Can also happen when using the value of variables

f is 1.5000 (the division is between a double and an integer, so divided as floating-point)

Accessing Variables

- Value of a variable can be accessed by name
- Wherever the variable name is used, the value stored in it is looked up and used
- Can be used anywhere a value is allowed

Recap

- Programs are a series of statements
- Defined in functions
- C programs start at the `main()` function
- Data stored in Variables
- Variables have *name* and *type*
- Statements can manipulate the variables

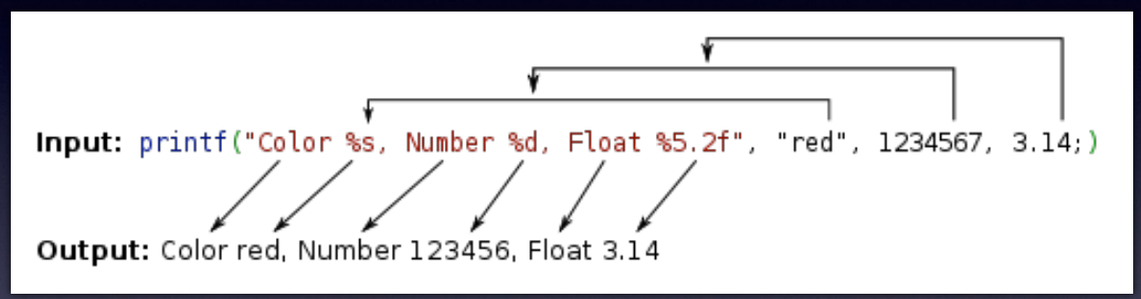
Printing

- Seen how to print text using `printf()`
- Note the '`f`'– stands for `print f`ormatted
- Takes one or more parameters
  - A string of text to print
  - A series of values to interpolate into that string

printf

- Regular characters are printed as normal
- Whenever a conversion specifier (e.g. `%d`) is encountered in the string
- The value of a parameter is interpolated into the string at that point
- In order…
- Value can be from anything…

Literal value, variable, function call etc...



Using real values here -- of course, we could also use the name of a variable as well

Interpolation String	`printf` meaning	Type
`%d`	print in decimal	`int`
`%i`	print in decimal	
`%o`	print in octal	
`%x/%X`	printf in hex	
`%u`	print unsigned	`unsigned int`
`%c`	print a single char	`int`

%c takes an int for printf (C will automatically promote a char to an int when you call printf)

Interpolation String	`printf` meaning	Type
`%s`	print string	`char *`
`%f`	print floating point number	`double`
`%lf`	print floating point number	`double`
`%e,%E`	print in exponent form	`double`
`%g, %G`	print appropriately	`double`
`%p`	print a pointer	`void *`
`%%`	print a '%'	`N/A`

string is a series of character that ends with \0
Give some demos...

Formatting

- Possible to be more specific about how things are formatted
- E.g. leading zeros, precision, etc…
- Expressed in the conversion specifier, e.g. `%5d` — always print 5 characters
- See `http://en.wikipedia.org/wiki/Printf_format_string`

Variable Scope

- Variables have *scope* – they can only be accessed from certain parts of the program
- So far, we've looked at *local* variables, only exist in that function
- Also possible to have *global* variables, global to the whole program

Local Variables

- Only accessible in the function defined in
- *Not* by functions called from that function
- Need to pass the value to the function as a parameter
- Note, the values of local variables don't persist between calls to a function

Go Demo this

Global Variables

- Aren't tied to a specific function
- Defined at the ‘top-level’ of the program outside of a function
- Can be accessed anywhere in the program
- Need to be declared before use
- Some people consider them bad practice

Think why you are making it global -- don’t just do it because it is easier
There should be a reason.

Parameters

- Parameters allow us to pass values from one function to another
- Work in a similar fashion to local variables, scope is limited to that function
- Value is assigned when the function is called
- Value can be changed just like a variable
- But the *original won't be changed!!!*

```
void PrintNumber(int number)
{
  printf(“Printing %d\n”, number);
}

int main(int argc, char *argv[])
{
  PrintNumber(42);
}
```

Go and demo this...

```
void PrintNumber(int number)
{
  printf(“Printing %d\n”, number);
}

int main(int argc, char *argv[])
{
  int value = 42;
  PrintNumber(value);
}
```

Go and demo this...

```
void PrintNumber(int number)
{
  printf(“Printing %d\n”, number);
  number = 56;
}

int main(int argc, char *argv[])
{
  int value = 42;
  PrintNumber(value);
  printf(“value still contains 42, look %d\n”,
       value);
}
```

Go and demo this...
Also demo it with two parameters

Global or local

- What happens if a global and local variable have the same name?
- This is fine
- But the function can only see and access the local variable (or parameter)
- Can’t have a local variable the same name as a parameter

Demo with DFB’s C example (page 8 on his notes)

return values

- Functions can return a value, as well take parameters
- Caller can use the returned value just like any other value
- Assign to a variable, pass as another parameter
- Or ignore it

return values

- Function uses the `return` keyword to return a value
- Stops the function at that point and returns to the caller passing the value back
- Can be anywhere in the function, not just the end

```
int LifeTheUniverseAndEverything()
{
  return 42;
}

int main(int argc, char *argv[])
{
  printf("The answer to life, etc. is %d\n",
    LifeTheUniverseAndEverything() );
}
```

Go and demo this...

```
double CelsiusToFahrenheit(double t)
{
  double celsius;

  celsius = 9.0 * t / 5.0 + 32.0;
  return celsius;
}
```

remember values have types...

Mathematical Operators

- C uses standard mathematical operators
- Work on one or two values (types will be promoted as necessary)
- These values can be from variables, functions, parameters etc.
- Compiler knows about *precedence* and *associativity*

Unary and Binary

- There are both unary and binary operators
- Unary operators bind more tightly than binary (but remember – can be both!)
- Operators of equal precedence bind left-to-right or right-to-left, depending on the operator

```
int ans = temp = 16 / 8 / 2;
```

means

```
int ans = (temp = (16 / 8) / 2);
```

`ans` and `temp` both contain `1`

/ binds left to right
= binds right to left

Precedence

- Unary operators bind tightest and are r-to-l
- Binary operators group left to right and have decreasing priority as follows

( )
* / %
+ -
>> <<
< > <= >=
== !=

Not a complete list

Assignment Precedence

- Assignment operators bind with the lowest priority
- Group right to left

=	+=	-=	>>=	<<=	…

Example

```
a = b + c * -d / e / f - g;
```

Example

```
a = b + c * -d / e / f - g;
```

- Unary minus on `-d` binds tightest

Example

```
a = b + c * -d / e / f - g;
```

- Unary minus on `-d` binds tightest
- `*` and `/` are equal but bind left-to-right

Example

```
a = b + (((c * (-d)) / e) / f) - g;
```

- Unary minus on `-d` binds tightest
- `*` and `/` are equal but bind left-to-right

Example

```
a = b + (((c * (-d)) / e) / f) - g;
```

- Unary minus on `-d` binds tightest
- `*` and `/` are equal but bind left-to-right
- Next, + and – are equal but lower priority and bind left-to-right so addition is done first

Example

```
a = (b + (((c * (-d)) / e) / f)) - g;
```

- Unary minus on `-d` binds tightest
- `*` and `/` are equal but bind left-to-right
- Next, + and – are equal but lower priority and bind left-to-right so addition is done first

Example

```
a = (b + (((c * (-d)) / e) / f)) - g;
```

- Unary minus on `-d` binds tightest
- `*` and `/` are equal but bind left-to-right
- Next, + and – are equal but lower priority and bind left-to-right so addition is done first
- Finally, the result is assigned to `a`