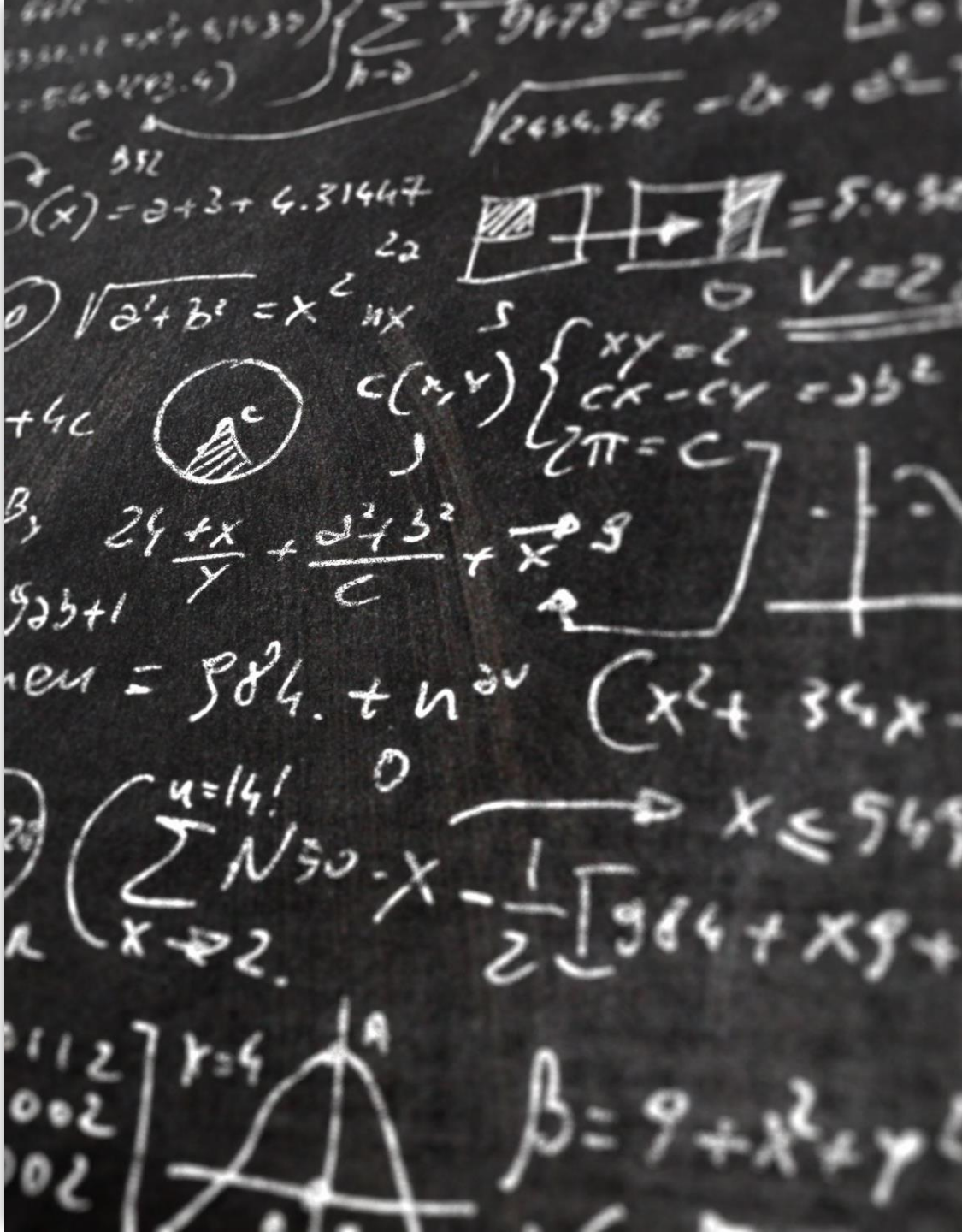
School of Computing & Communications | Lancaster University

SCC.111 Software Development – Lecture 5: Variables & Arrays

Adrian Friday, Nigel Davies, Hansi Hettiarachchi, Saad Ezzini

This lecture

- More on variables and storage (introducing more data types & arrays)
- What is a variable's 'scope' and where should you put them
- Example of how data flows and scope on data visibility



Critical to programming is ‘representing things’ as data

- It’s the ‘model’ of the world we manipulate
  - Samples of audio
  - Patient health records
  - Positions of virtual alien spaceships
  - Graphics
  - The status of computations
  - Machine learning models
  - Web pages

C is a **typed** language

Variables, function parameters and return values all have a ‘type’. We often have to convert from one representation to another

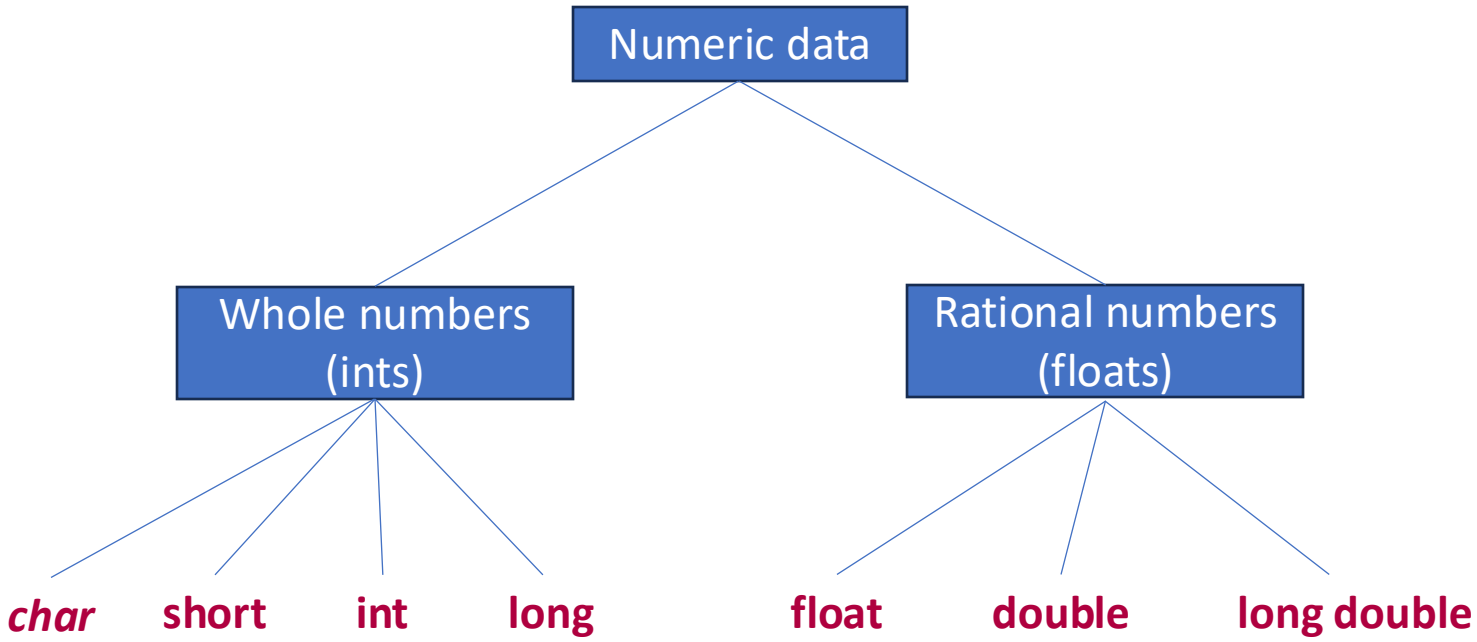
Types vary from one another

C has many basic types

Integer or Rational	Range of values	Signed or unsigned	Precision

We pick ‘the best type’ for our application

We store data in variables of **various types** which constraints their **size** in memory and **precision**



42

42.0

We use different constant ‘literals’, which also have ‘type’

e.g. `long bigInt = 0xFeeL;`

```
1
212 /* Legal */
0213 /* octal, base 8 */
0x4b /* hexadecimal, base 16 */
0xFeeL /* Legal, base 16, long int ☺ */
215u /* Legal, unsigned int */
078 /* Illegal: 8 is not an octal digit, only 0-7 */
3.14159 /* Legal, float */
314159E-5L /* Legal, float, scientific notation */
```

But literals often need ‘type conversion’

```
int main()
{
  int x = 42.0;
  printf("%d\n", x);

  return 0;
}
```

What happens when a floating point literal becomes an integer?

Variables have a ‘scope’

Variables need to be ‘in the right scope’ to be visible where they are needed

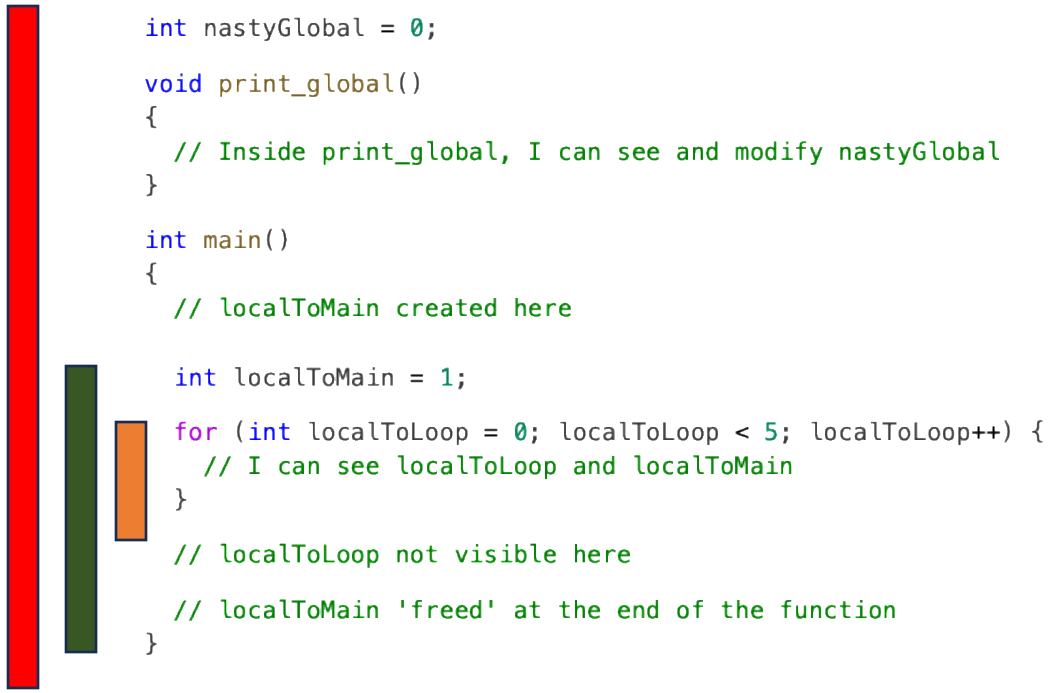
x is ‘local’ to the function ‘main’

```
int main()
{
  int x = 42.0;

  printf("%d\n", x);

  return 0;
}
```

It is **created** when it is declared, and is automatically **destroyed** when the function ends

Variables are visible at the scope where they are created

- A variable exists in the code block where it is declared
  - Local to functions
  - Local to blocks {} (e.g. the body of an if or while)
  - Global variables (which we try never to use!) are visible at 'whole file' scope



```
int nastyGlobal = 0;

void print_global()
{
  // Inside print_global, I can see and modify nastyGlobal
}

int main()
{
  // localToMain created here

  int localToMain = 1;

  for (int localToLoop = 0; localToLoop < 5; localToLoop++) {
    // I can see localToLoop and localToMain
  }

  // localToLoop not visible here

  // localToMain 'freed' at the end of the function
}
```

Basic rules about variables

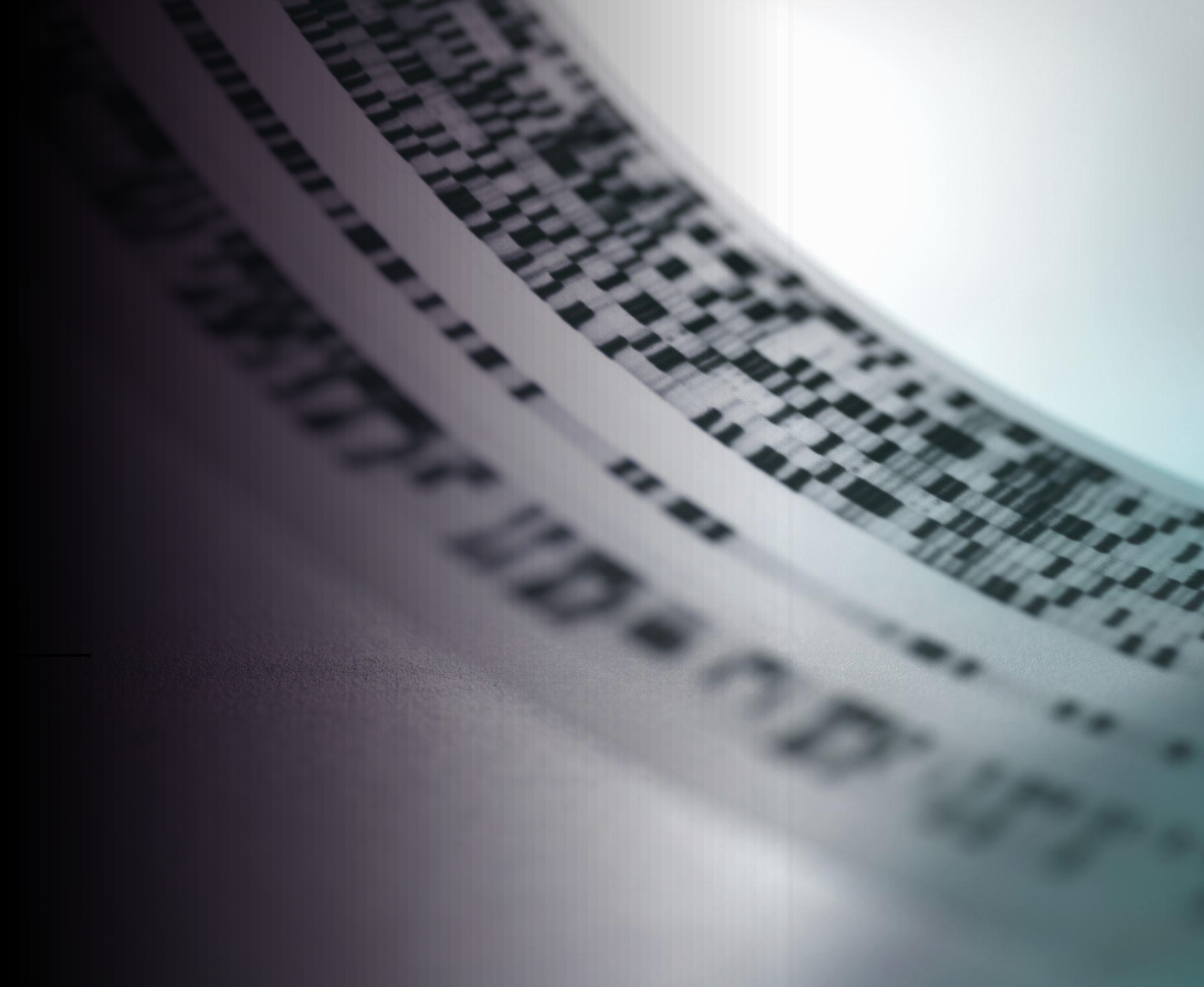
A variable **must be** declared before it is used

Generally, we declare variables '**close**' to where you need to use them

A variable *should be initialised* with some value before it is used also

Code walkthrough

Variable scope and Q&A



For reference

```
int nastyGlobal = 0;

void print_global()
{
  // Inside print_global, I can see and modify nastyGlobal
}

int main()
{
  // localToMain created here

  int localToMain = 1;

  for (int localToLoop = 0; localToLoop < 5; localToLoop++) {
    // I can see localToLoop and localToMain
  }

  // localToLoop not visible here

  // localToMain 'freed' at the end of the function
}
```

So, if variables are bound to their scope…

- How do we get the data where we need it in other scopes?
  - Global variables? *Definitely not!*
  - We need to pass data as parameters to functions
  - And return results using ‘return’ so functions evaluate to something (*we often assign the result of a function call into a variable to use it later*)

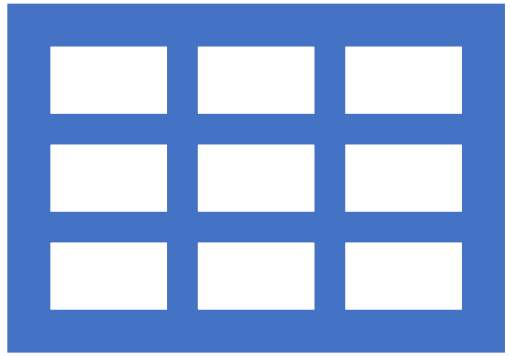
Data ‘flows’ via parameter passing into functions and return values

```
int main()
{
  int userInput;

  scanf("%d", &userInput);

  print_user_input(userInput);
}
```

```
void print_user_input(int dataInHere)
{
  // dataInHere is a parameter
  // it has 'local scope' to this function

  printf("User input was %d\n",
        dataInHere);
}
```

What if we want to pass sequences of things (as **arrays**)?

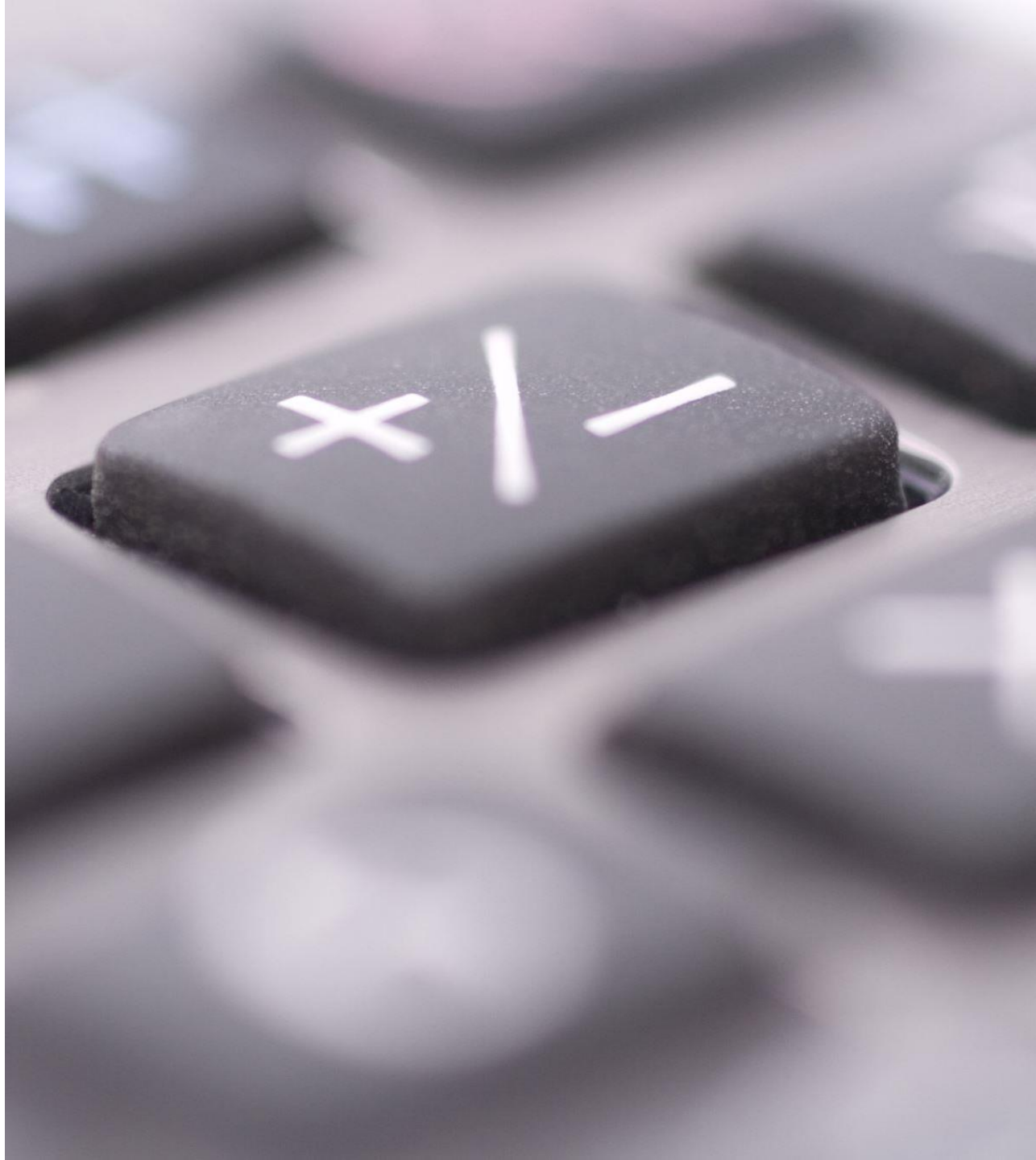
- Consider this task
  - Add up a list of integer numbers and display the total
  - How should we represent this as a variable data type?
  - What questions do you have already about this specification?



We often need to be able to store and manage bigger blocks of data – for this we **use arrays**

We need a way to represent many similar items

Arrays provide a simple way of storing multiple instances ‘a sequence’ of *the same* data type



We can create (declare) arrays using []

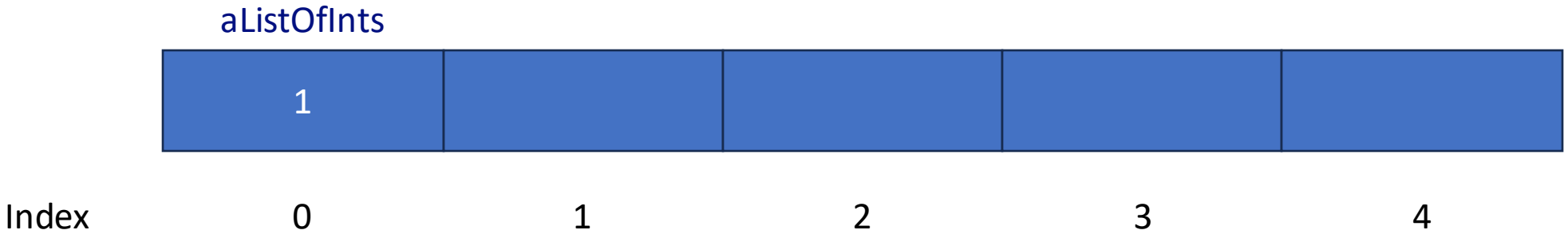
- int aListOfInts[5]; // array of 5 integers

aListOfInts

Note: C supports up to 7 dimensional arrays, see K&R for more details

Arrays are indexed from 0

- aListOfInts[0] = 1; // make 1st element 1
- int x = aListOfInts[0]; // assign x



We can also initialise arrays like this

- int aListOfInts[] = { 1, 2, 3, 4, 5 };

aListOfInts

1	2	3	4	5

Note: if we provide an initialiser we can initialise only part of the array

int aListOfInts[10] = { 1, 2, 3, 4, 5 };

Indexing arrays

- int aListOfInts[5] = { 1, 2, 3, 4, 5 };
- aListOfInts[0] = 5; // overwriting the 1st

aListOfInts

5	2	3	4	5

```
printf("Element 3 is %d\n",
        aListOfInts[2]);
```

Note: indexes start at 0, so an array with 5 elements is indexed 0..4 inclusive

Arrays are just named contiguous data blocks in memory (*no frills!*)!

- They do not know how many elements are initialised
- They will not grow if you access memory before or after the allocated space!
- The C compiler will **not protect you** from exceeding the array bounds!
- You will need to track this yourself and *code defensively*

```
int aListOfInts[] = { 1, 2, 3, 4, 5 };

printf("Element 5 is %d\n",
      aListOfInts[5]); // out of bounds

aListOfInts[5] = 6; // probably crashes!


if (index >= 0 && index < 5)
  aListOfInts[index] = 6; // should be ok
```

Progress so far

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

Summary

- You should know about literals and variable data types
- Scopes and the importance of where variables are declared
- You should know how to declare and index into arrays to store/retrieve lists of values
- Arrays are indexed from 0, take care not to go ‘out of bounds’