Lecture 6 Functions

$$f(x) = a + bx + cx^2$$

Functions

- Readability – a program divided into small pieces are easier to understand and modify.
- Modulization – isolate the small pieces by allowing a few explicit connection for clarity.
- Simplicity – avoid duplicating code to be used more than once.
- Reusability – code for certain purposes can be reused in other programs.

Defining a function

Return type

Can *not* be an array!

Name of the function

Arguments

May contain arrays

```
double average(double a, double b)
{
    return (a + b) / 2;
}
```

Body of the function

Return statement
(match the return type)

Body of a function

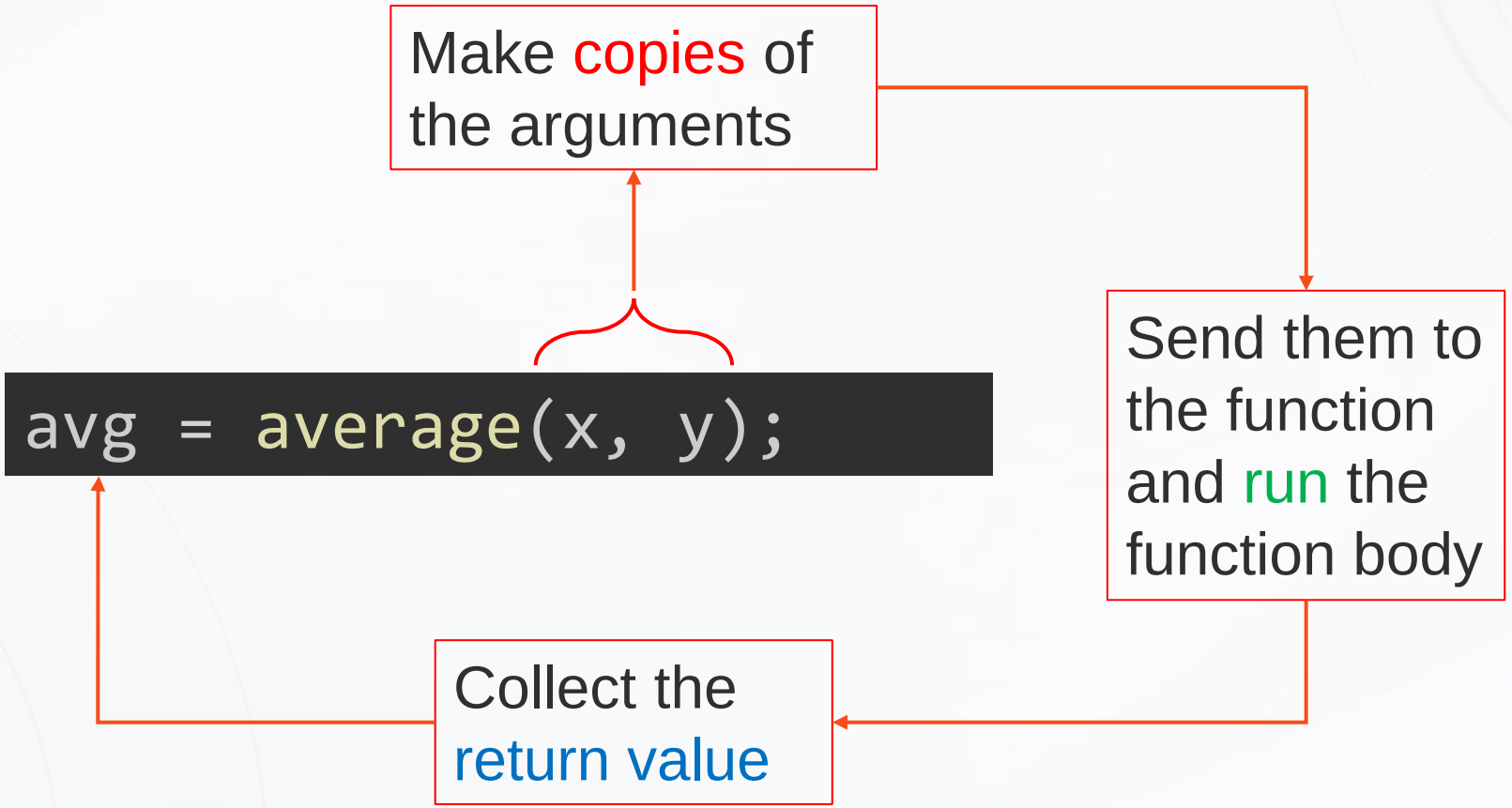
May declare local variables

valid within the block, after declaration

May contain multiple statements

```
double average(double a, double b)
{
    double sum;          /* declaration */
    sum = a + b;         /* statement */
    return sum / 2;      /* statement */
}
```

Calling a function



```c
#include <stdio.h>

double average(double a, double b)
{
    return (a + b) / 2;
}

int main(void)
{
    double x, y, z;

    printf("Enter three numbers: ");
    scanf("%lf%lf%lf", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    printf("Average of %g and %g: %g\n", y, z, average(y, z));
    printf("Average of %g and %g: %g\n", x, z, average(x, z));
    return 0;
}
```

A function with no return value

Void type

```
void print_count(int n)
{
    printf("T minus %d and counting\n", n);
}
```

No need for a return statement

- To call the function

```
int main(void)
{
    for (int i = 10; i > 0; --i)
        print_count(i);
    return 0;
}
```

Don’t attempt to obtain a return value

A function with no parameter

The void here indicates that there is no parameter

```
void print_pun(void)
{
    printf("To C, or not to C: that is the question.\n");
}
```

- To call the function

```
    print_pun();
```

No input, but the parentheses must be present.

A return value can be discarded

```c
#include <stdio.h>

int main(void)
{
    int ret_val;
    ret_val = printf("Hello world!\n");
    printf("The return value is: %d", ret_val);
}
```

To explicitly discard the return value

```c
    (void) printf("Hello world!\n");
```

Just to tell others that you intentionally discard the return value.

Example: Test whether a number is prime

```
Enter a number: 34
Not prime
```

The program uses a function named `is_prime` that returns `true` if its parameter is a prime number and `false` if it isn't.

Definition of prime numbers?

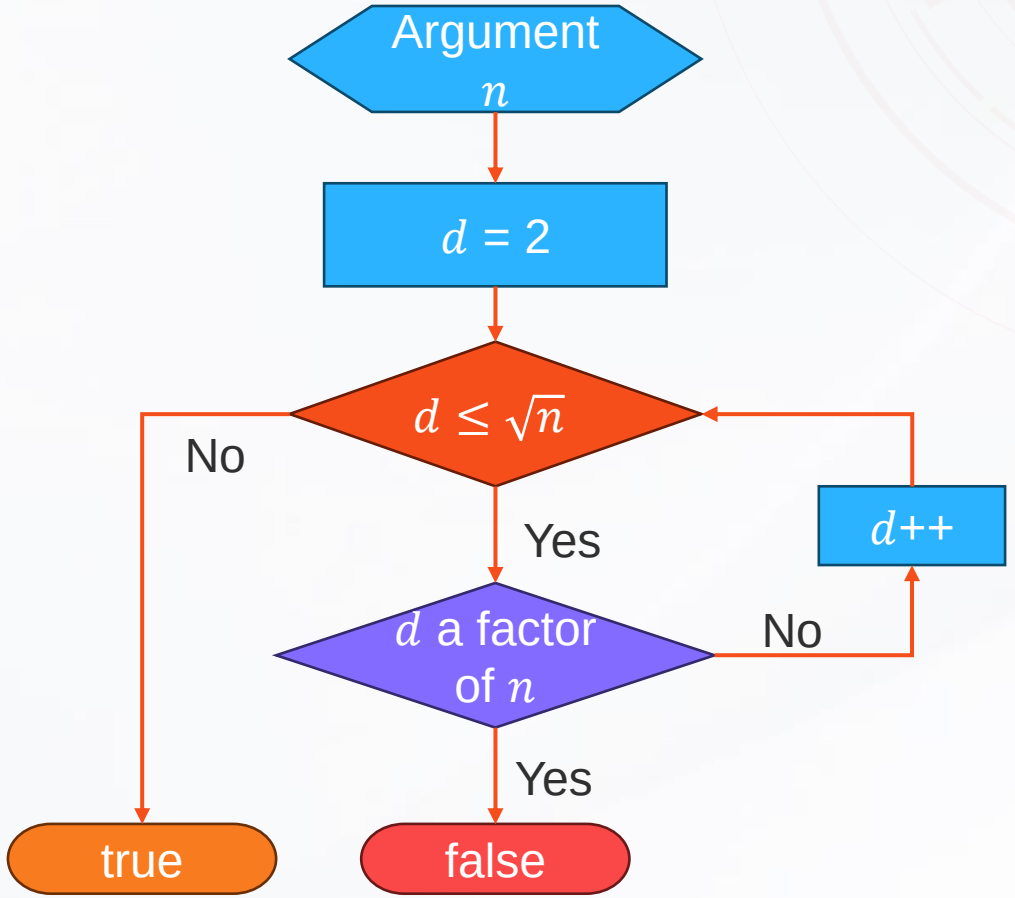
is_prime function

A prime number: no other factors except 1 and itself

Try all factors from 2 to ~~n-1~~ $\sqrt{n}$

How to determine whether d is a factor of n?

n % d == 0



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

bool is_prime(int n)
{
    if (n <= 1)
        return false;
    for (int d = 2; d*d <= n; d++)
        if (n % d == 0) return false;
    return true;
}
```

```c
int main(void)
{
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    if (is_prime(n))
        printf("Prime\n");
    else
        printf("Not prime\n");
    return 0;
}
```

Can we place a function after main()?

```c
#include <stdio.h>

int main(void)
{
    double x, y, z;
    printf("Enter three numbers: ");
    scanf("%lf%lf%lf", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    printf("Average of %g and %g: %g\n", y, z, average(y, z));
    printf("Average of %g and %g: %g\n", x, z, average(x, z));
    return 0;
}

double average(double a, double b)
{
    return (a + b) / 2;
}
```

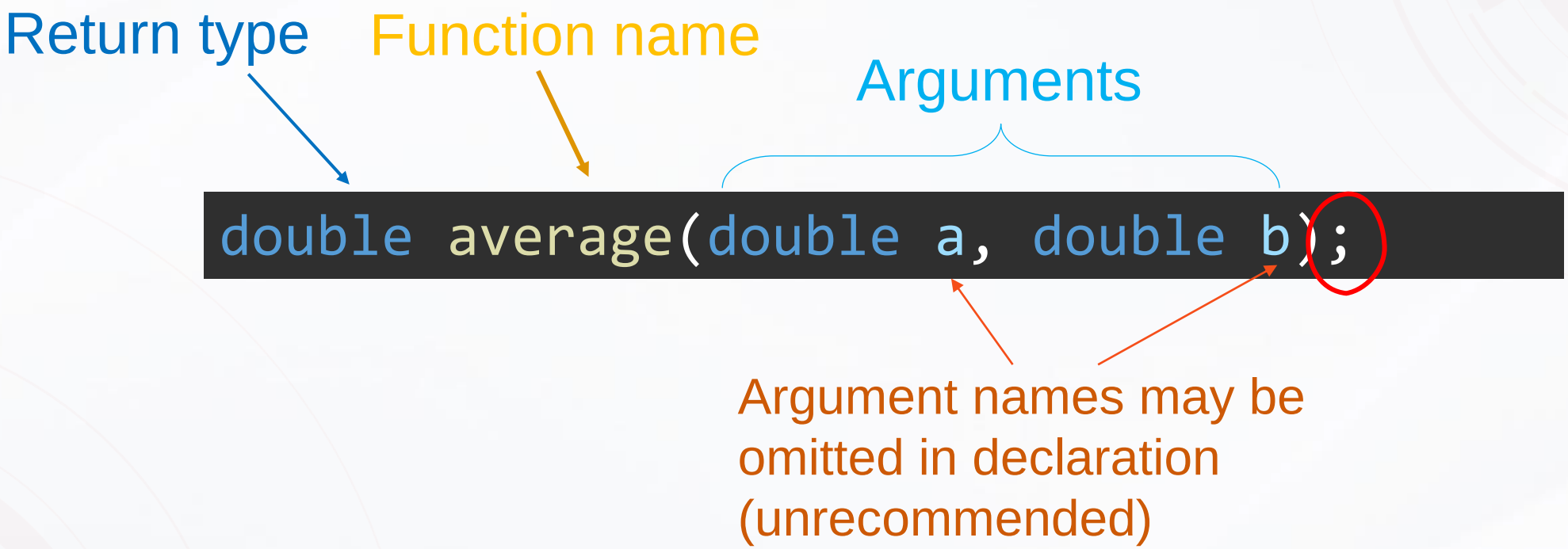
Problems with implicit declaration

- An int return value is always assumed.
- The compiler is unable to check whether correct number & type of parameters are passed in.

~~Define~~ before usage!

Declare

Declaring a function



Declare before usage!

```c
#include <stdio.h>          /* Declaration of printf & scanf included */

double average(double a, double b);   /* Declaration of average */
int main(void)
{
    double x, y, z;
    printf("Enter three numbers: ");
    scanf("%lf%lf%lf", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    printf("Average of %g and %g: %g\n", y, z, average(y, z));
    printf("Average of %g and %g: %g\n", x, z, average(x, z));
    return 0;
}

double average(double a, double b)    /* Implementation of average */
{
    return (a + b) / 2;
}
```

Function arguments are passed by value

- Copies of the arguments are made before passing into the function.
- Changes made over the arguments in a function won't affect original ones.

```
void decompose(double x, long int_part,
               double frac_part)
{
    int_part = (long) x;
    frac_part = x - int_part;
}
```

What's wrong?

Use pointers to allow changes over arguments

```
void decompose(double x, long *int_part,
               double *frac_part)
{
    *int_part = (long) x;
    *frac_part = x - *int_part;
}
```

Array arguments

- When a function parameter is a one-dimensional array, the length of the array can be left unspecified (*pointer decay*):

```
int f(int a[])  /* no length specified */
{
        …
}
```

- C doesn't provide any easy way for a function to determine the length of an array passed to it.
- Instead, we'll have to supply the length—if the function needs it—as an additional argument.

Length of array passed in as additional argument

```
int sum_array(int a[], int n)
{
    int i, sum = 0;

    for (i = 0; i < n; i++)
     sum += a[i];

    return sum;
}
```

Even simpler/clearer: variable length array (C99+)

Define before using it as a size

```
int sum_array(int n, int a[n])
{
    int i, sum = 0;

    for (i = 0; i < n; i++)
    sum += a[i];

    return sum;
}
```

Note: the actual length of the array passed in *can* be different.

```c
#define LEN 100

int sum_array(int n, int a[*]);

int main(void)
{
    int b[LEN], total;
    …
    total = sum_array(50, b);
    …
}
```

You may use n or * in declaration, to indicate that this is a variable length array.

Summing over the first 50 elements of b.

Warning: C does *not* check the bounds of an array.

Elements of an array argument may be changed

A function to fill the (first n) elements of an array with zeros.

```
void zeros(int n, int a[n])
{
    for (int i = 0; i < n; i++)
        a[i] = 0;
}
```

Multidimensional array arguments

Conventional array argument: You must specify all other (inner) dimensions, only the length of the first dimension may be omitted.

```
#define LEN 10

int sum_2D_array(int a[][LEN], int n)
{
    int i, j, sum = 0;

    for (i = 0; i < n; i++)
        for (j = 0; j < LEN; j++)
            sum += a[i][j];

    return sum;
}
```

Variable length arrays

```
int sum_2D_array(int n, int m, int a[n][m])
{
    int i, j, sum = 0;

    for (i = 0; i < n; i++)
        for (j = 0; j < LEN; j++)
            sum += a[i][j];

    return sum;
}
```

Compound literals as input arguments

```
int b[] = {3, 0, 3, 4, 1};
total = sum_array(5, b);
```

If b isn't needed for any other purpose, these statements can be slightly simplified

```
total = sum_array(5, (int []){3, 0, 3, 4, 1});
```

Compound literal creates an array "*on the fly*".

You may specify the number of elements, and omit the zeros.

```
(int [10]){3, 6}
```

return statement

- A non-`void` function must use the `return` statement to specify what value it will return.

  `return` *expression*`;`

```
return 0;
```

```
return sum;
```

```
return n >= 0 ? n : 0;
```

- *Implicit conversion* if type does not match

Implicit conversion

- When the operands in an arithmetic or logical expression don't have the same type.
- When the right side of an assignment doesn't match the type of the variable on the left side.
- When the type of an argument in a function call doesn't match the type in the definition/declaration.
- When the type of the expression in a `return` statement doesn't match the function's return type.