CS2311 Computer Programming

LT7: Functions

Outline

- Function declaration
- Parameter passing, return value
- Passing an array to a function
- Function Prototype
- Recursive functions

What is function?

- A collection of statements that perform a specific task.

- Functions are used to break a problem down into manageable pieces
  - KISS principle: "Keep it simple, Stupid!"
  - Break the problem down into small functions, each does only one simple task, and does it correctly

- A function can be invoked multiple times. No need to repeat the same code in multiple parts of the program.

Structured Programming Guidelines

- Flow of control in a program should be as simple as possible
- Construction of program should embody a top-down design
  - Decompose a problem into small problems repeatedly until they are simple enough to be coded easily
  - From another perspective, each problem can be viewed from different levels of abstraction (or details)
  - Top-down approach of problem solving is well exercised by human beings

Function in C++

- The C++ standard library provides a rich collection of functions
- Mathematical calculations (`#include <cmath>`)
- String manipulations (`#include <cstring>`)
  - `#include <string>`
- Input/output (`#include <iostream>`)
- Some functions are defined in multiple library in some platform, e.g. function `sqrt` is defined in both `cmath` and `iostream` in VS

How to use a function written by others?

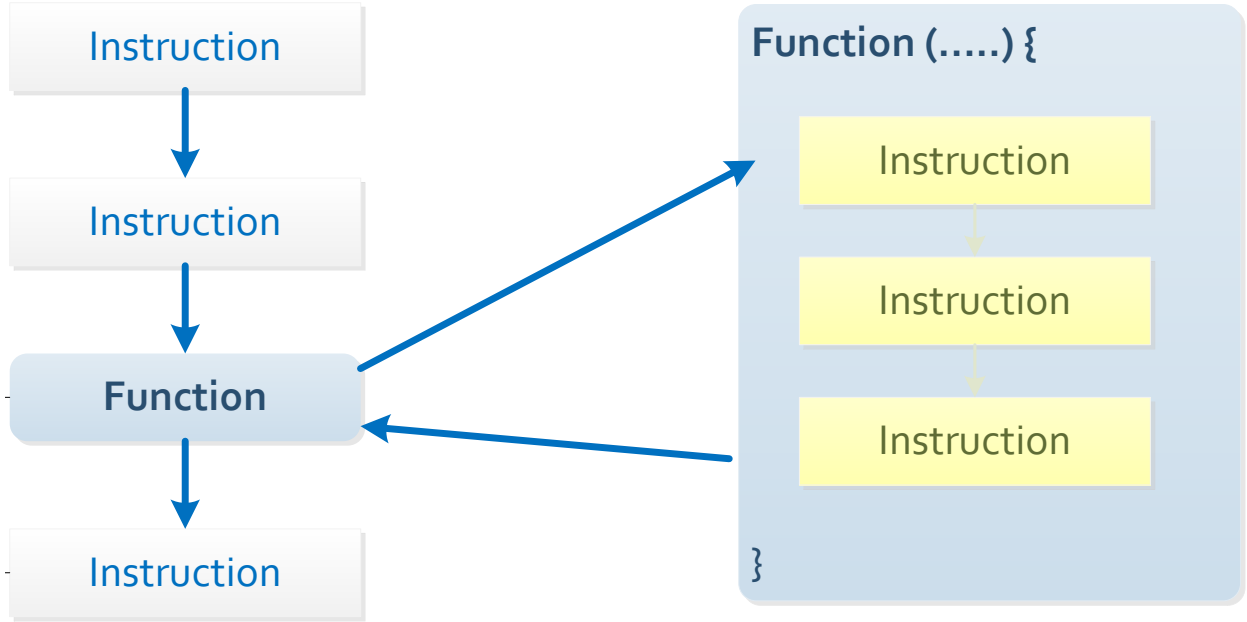
Tell compiler that you are going to use functions defined in **iostream** package

```
#include <iostream>
using namespace std;

int main() {
  double area, side;
  cout << "Enter the area of a square: ";
  cin >> area;
  side = sqrt(area);
  cout << "The square has perimeter: " << 4 * side << endl;
  return 0;
}
```

Pass area to the function **sqrt** which will return the square root of **area**

Function Invocation

- During program execution, when a function name followed by parentheses is encountered, the function is invoked and the program control is passed to that function; when the function ends, program control is returned to the statement immediately after the function call in the original function

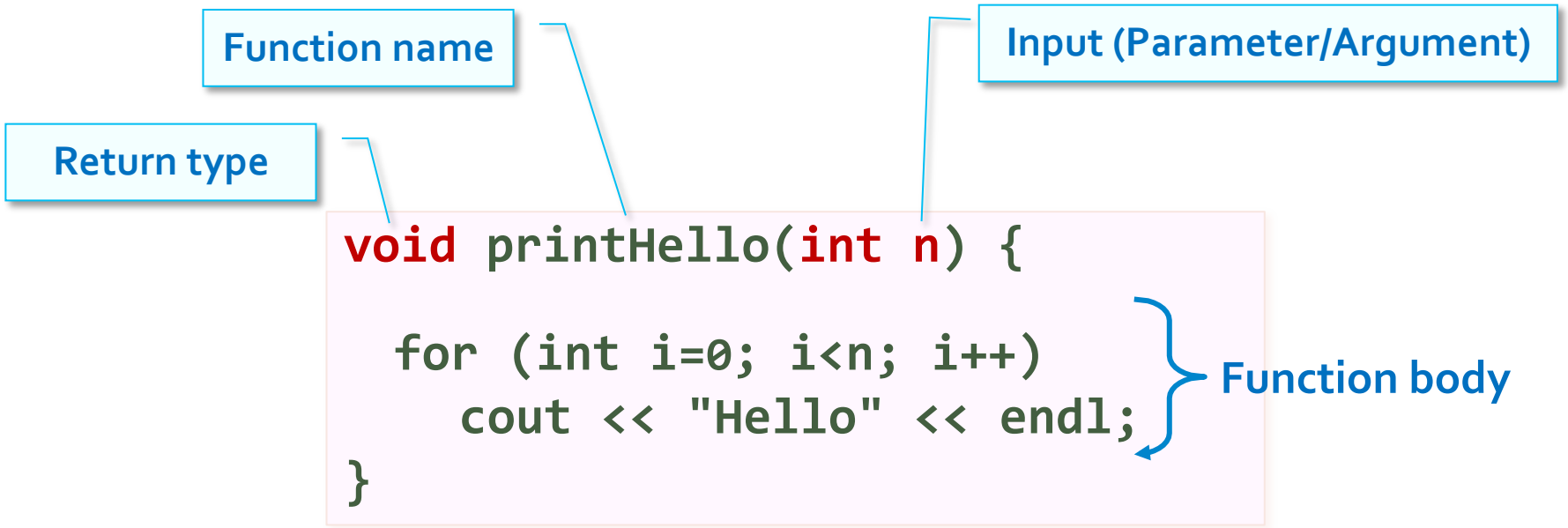


Write Your Own function
(User defined functions)

- Define a function **`printHello`** which accepts an integer `n` as input

- The function should print "**`Hello`**" `n` times, where `n` is an integer

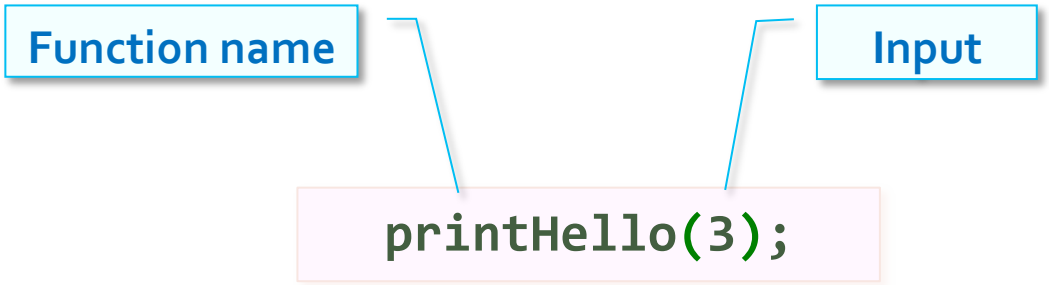
Function Components



n is defined as input,
therefore there is no need to declare **n** in the function body again

Calling / Invoking a function (I)

To make a function call, we only need to specify a function name and provide parameters in a pair of `()`



We don't need the return type when calling a function.

Syntax error:

```
int printHello(3);
```

Calling / Invoking a function (II)

```
int x = 4;
printHello(x);
printHello(x+2);
```

Print "hello" 4 times and then 6 times.
We don't need the parameter type when calling a function.

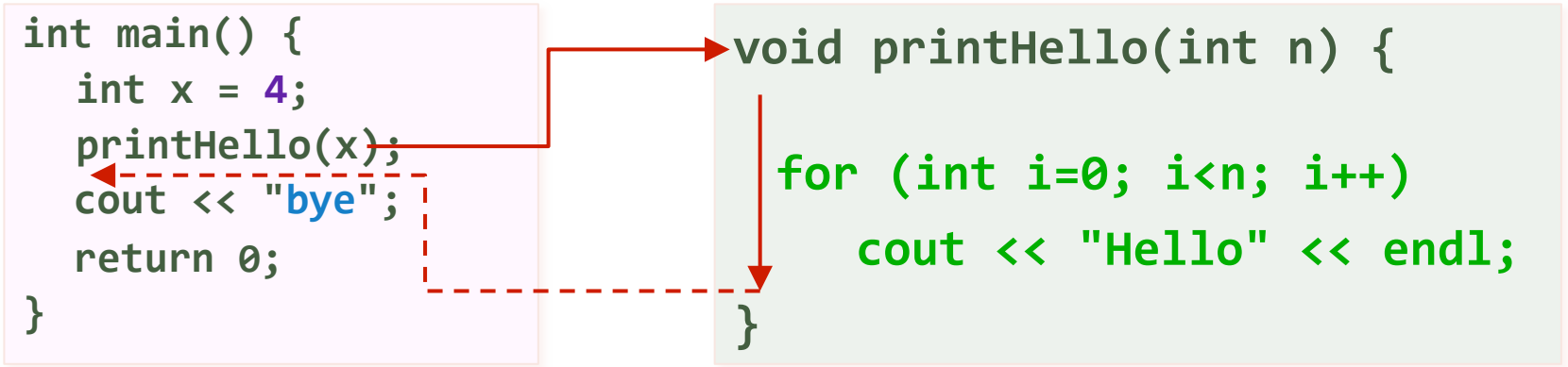
Syntax error:

```
printHello(int x);
```

Advantage of using a function:

we don't need to write two loops, one to print "hello" 4 times and the other to print "hello" 6 times

Flow of Control



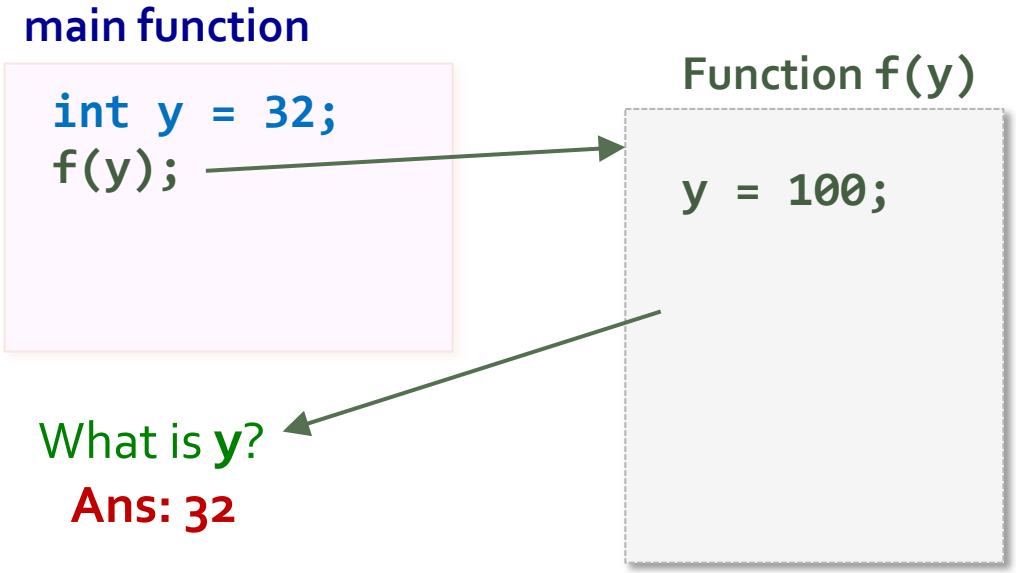
1. The program first start execution in `main()`
2. `printHello(x)` is called
3. The value of `x` is copied to the variable `n`. As a result, `n` gets a value of `4`.
4. The loop body is executed.
5. After executing all the statements within `printHello()`, control go back to `main()` and "bye" is printed

Function with No Input

```
void printHello() {
    cout << "hello";
}
```

Parameter Passing: Call-by-Value

- When a function is invoked, the arguments within the parentheses are passed using **call-by-value**
- Each argument is evaluated, and its value is used locally in place of the corresponding parameter



Function Variables

LOCAL TO THE FUNCTION

Parameters Passing: Call-by-Value

```
void f(int x) {
   x = 4;          // we modify the value x to 4
   // Do we modify y at the same time? NO
   y = 4;          // syntax error: y is local to main
}
int main() {
   int y = 3;
   f(y);
   cout << y;      // print 3, y remains unchanged
   return 0;
}
```

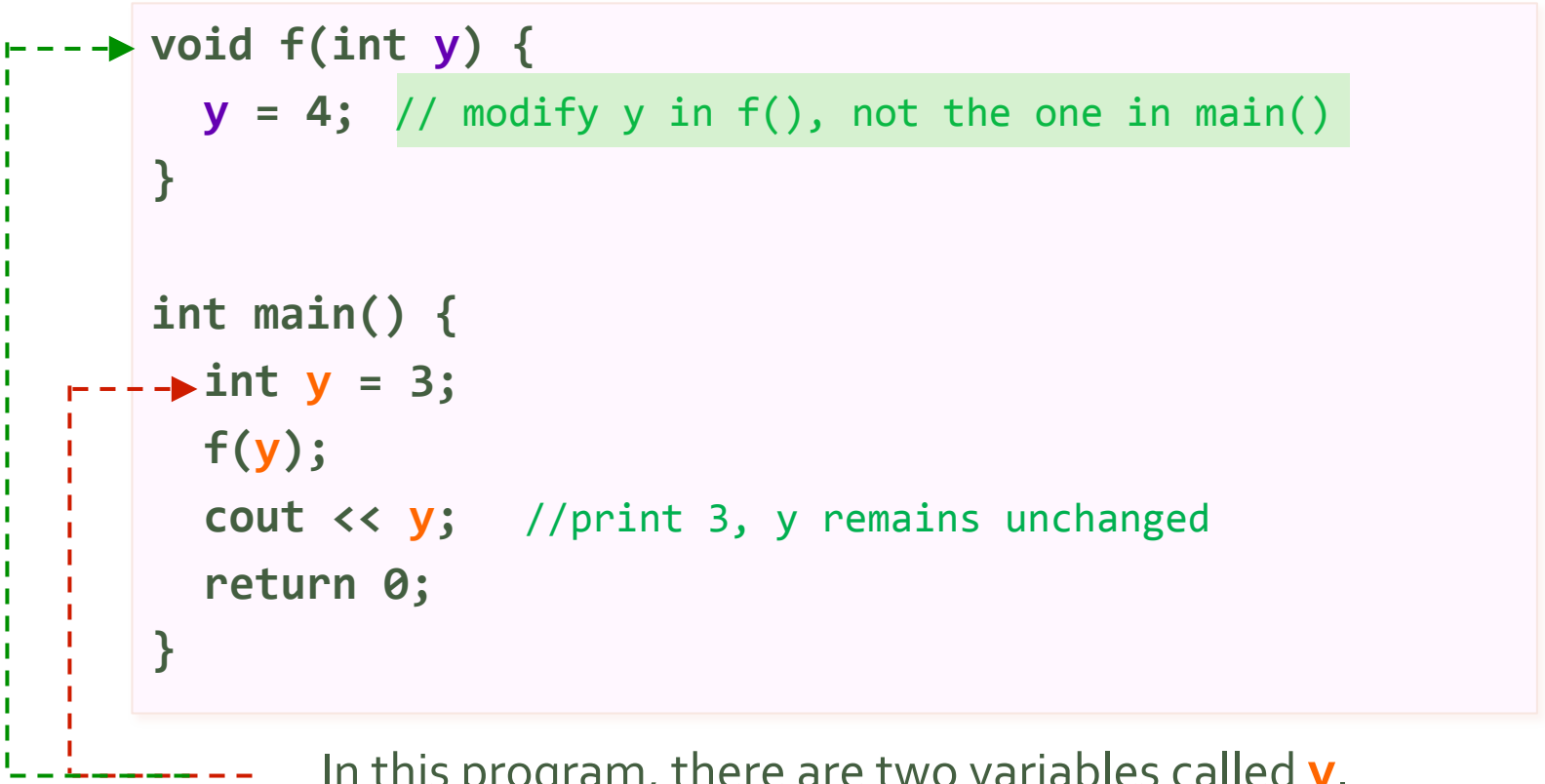
The variables **x** and **y** are **local** variables.

y is local to **main()**, so we cannot use **y** in **f()**.

x and **y** are two independent variables.

When **x** is modified, **y** will not be affected.

What if We Change x to y in f()?

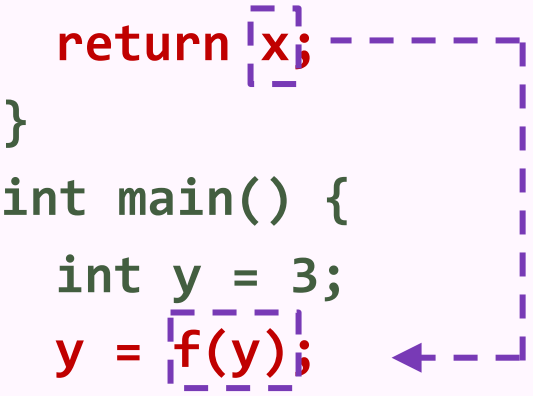


```
void f(int y) {
  y = 4;  // modify y in f(), not the one in main()
}

int main() {
  int y = 3;
  f(y);
  cout << y;   //print 3, y remains unchanged
  return 0;
}
```

In this program, there are two variables called **y**.
One is defined in **f()** and one is defined in **main()**.
In **f()**, **y** in **f()** is modified.
However, **y** in **main()** is not affected.

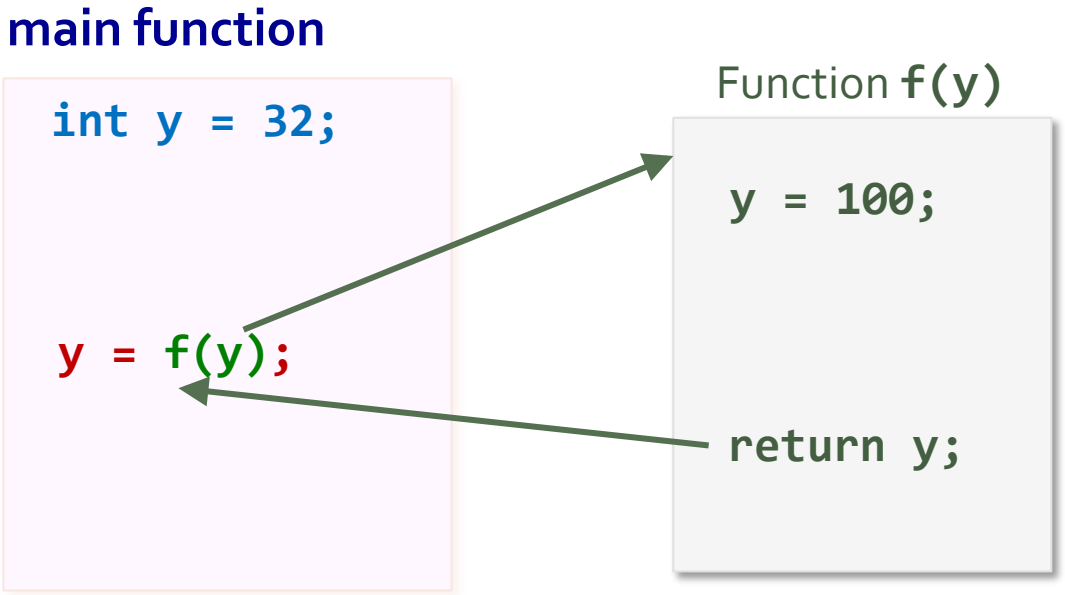
How to Modify y in f()?

```
int f(int x) {
  x = 4;  // we modify the value x to 4
  // Do we modify y at the same time? NO
  return x;
}
int main() {
  int y = 3;
  y = f(y);
  cout << y;     // print 4
  return 0;
}
```



By assigning the return value of **f(y)** to **y**,
after the function call,
y gets a value of **4**

Return Value



We assign the return value of **f(y)** to the variable **y**.
What is **y**?
Ans: **100**

The **return** Statement

- When a **`return`** is encountered, the value of the (optional) expression after the keyword return is sent back to the calling function
- The returning value of a function will be converted implicitly, if necessary, to the type specified in the function definition
- Syntax:

```
return expression;
return;
```

  - Example:

```
return (a+b*2);
```

Examples: Function with return value

Function	Parameter	return value	Examples:

getX	nil	int	`int getX() {`
			`  int d;`
			`  cin >> d;`
			`  return d;`
			`}`

calMax	double f1
	double f2
		double	`double calMax(double f1,double f2) {`
			`  if (f1 > f2)`
			`  return f1;`
			`  else`
			`  return f2;`
			`}`

getInput	int n1,
	double n2
		char	`char getInput(int n1, double n2) {`
			`  char c;`
			
			`  return c;`
			`}`

Examples: Function w/o return value

Function	Parameter	return value	Examples:

printHello	nil	nil	`void printHello() {`
			`  cout << "Hello\n";`
			`}`

printHellos	int n	nil	`void printHellos(int n) {`
			`  for (int i=0; i<n; i++)`
			`  cout << "Hello\n";`
			`}`

printMax	float n1
	float n2
		nil	`void printMax(float n1, float n2) {`
			`  cout << ((n1>n2)?n1:n2) << endl;`
			`}`

printFloats	int n
	float data
		nil	`void printFloats(int n, float data) {`
			`  …`
			`}`
			`or`
			`void printFloats(float data, int n) {`
			`  …`
			`}`

Example: `findMax`

- We can define a function **`findMax`**, which accepts two integers as input.
  - The function **returns** the larger value of the two integers.
  - E.g.

    When **`x > y`**, the expression **`findMax(x, y)`** should evaluate to a value of **`x`**

```
cout << findMax(4, 3); //print x (4)
```

    When **`y > x`**, the expression **`findMax(x, y)`** should evaluate to a value of **`y`**

```
cout << findMax(3, 4); //print y (4)
```

Function Implementation

```
int findMax(int n1, int n2) {

  if (n1 > n2)
     return n1;
  else
     return n2;
}
```

The return type of the variable is `int`.
When there are more than one arguments,
they are separated by a comma.
The type of each variable should be specified individually.

Error:

```
int findMax(int a, b);
```

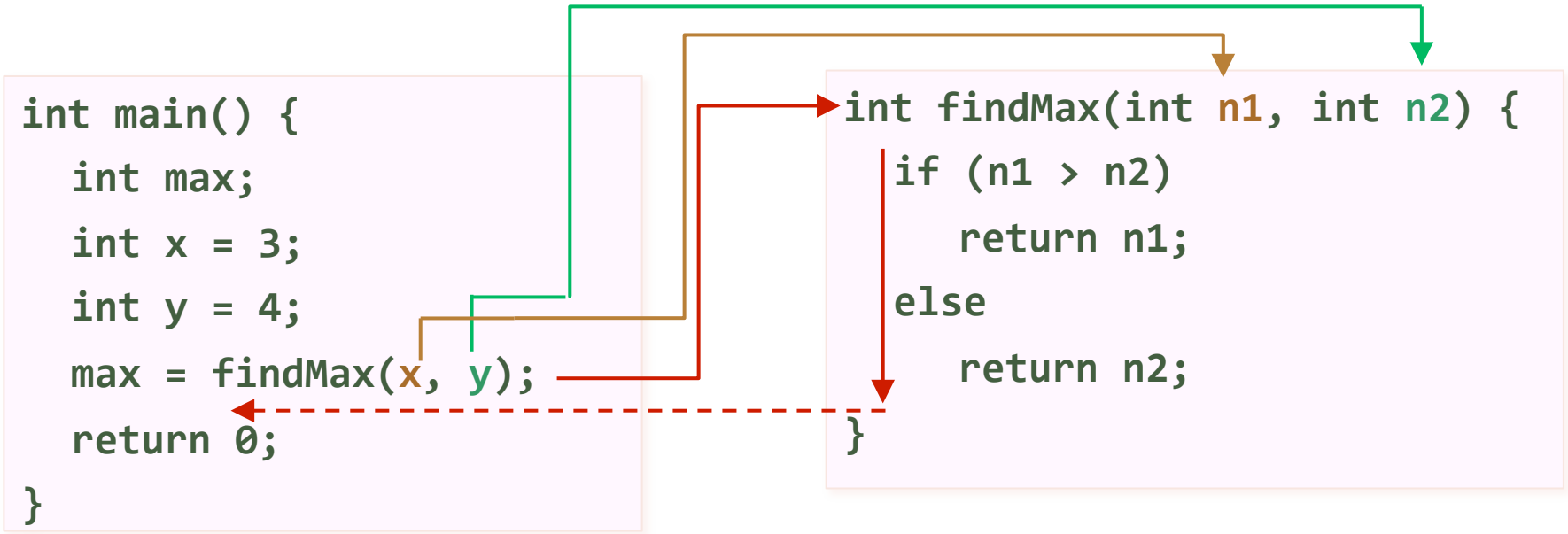
Calling `findMax()`

```
int max;
int x = 3;
int y = 4;
max = findMax(x, y);
```

The value of this expression is 4. Assign 4 to **`max`**.

The variable **`max`** will hold the value of **`x`** when **`x>y`**.
Otherwise, **`max`** will hold the value of **`y`**.

Flow of Control



When **findMax()** is called,
the value of **x(3)** is copied to the variable **n1**
the value of **y(4)** is copied to the variable **n2**

Finding the max of 3 numbers, i, j, k

```
int i, j, k;
int max;
cin >> i >> j >> k;

//find the max of i, j, k
___ = findMax (___ , ___);
___ = findMax (___ , ___);

cout << "max is " << max;
```

Answer

```
int i,j,k;
int max;
cin >> i >> j >>k;

// find the max of i, j, k
max = findMax(i, j); // max stores 4
max = findMax(max, k); // max stores 6
cout <<"max is " << max;
```

```
3 4 6
max is 6
```

What is the Output of the Following Program?

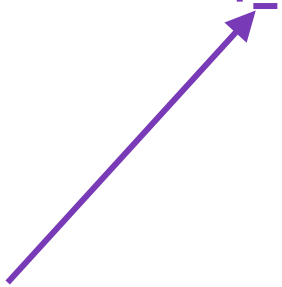
```
void f(int y, int x) {
  cout << "x =" << x << endl;
  cout << "y =" << y << endl;
}

int main() {
  int x = 3, y = 4;
  f(x, y);
  return 0;
}
```

```
x = 4
y = 3
```

Parameter Passing: Default Parameters

- We can also provide some *default* values for certain parameters
- Example:

```
void f(int a, int b, int c = 0) {
    …
}
```



Default parameter with a default value

Parameter Passing: Default Parameters

- If no value is passed to the default parameter, the complier will use its *default* value in the function call
- Example

```
void f(int a, int b, int c = 0) {
    …
}
int main() {
    f(3, 4);     // c = 0 in the function call
    f(3, 4, 5); // c = 5 in the function call
    return 0;
}
```

Parameter Passing: Default Parameters

- All the default parameters must locate at the *right-hand side* of normal parameters
- Invalid examples:

```
void f(int a, int b = 0, int c, int d = 0){
    // invalid definition, the default parameter b
    // locates at left-hand side of c
}

void f(int a, int b, int c = 0, int d = 0){
    // valid definition
}
```

Parameters Passing: Arrays

- When passing an array to a function, we only need to specify the array name
- The following example is invalid

```
void f(int x[20]) {
    …
}

int main() {
    int y[20];
    f(y[0]);    //invalid, type mismatch
    return 0;
}
```

this is an array of int

this is an int

Parameters Passing: Arrays

The size of array is optional.
`void f(int a[])`

if the content of **a[i]** is modified in the function, the modification will persist even after the function returns (Call by reference)

```
void f(int a[3]) {
   cout << a[3] << endl; //1 is printed
   a[0] = 10;
}
int main () {
   int a[3] = {1, 2, 5}; //an array with 3 elements
   f(a);                 //calling f() with array a
   cout << a[0] << endl; //10 is printed
   return 0;
}
```

Only need to input the array name!

Parameter Passing: 2D array

- The way to pass a 2D array is similar as the 1D array
- Example:
  - define a function which reads a 2D array as the input and *sort each row of the input 2D array*

```
void sort2D(int x[][10]){
   …
}

int main() {
   int y[20][10];
   sort2D(y);
   return 0;
}
```

The size of the **first** dimension is **optional**, while the size of the **second** dimension **must be given**

Example: Sort rows of 2D arrays

```
void sort2D(int a[][3]) {
   int tmp;
   for (int i = 0; i < 3; i++)          // each row
      for (int j = 0; j < 3 - 1; j++)  // bubble sort
         for (int k = 3 - 1; k > j; k--)
            if (a[i][k] < a[i][k - 1]) {
               tmp = a[i][k];          // swap neighbors
               a[i][k] = a[i][k - 1];
               a[i][k - 1] = tmp;
            }
}
int main() {
   int a[3][3] = {0};
   for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
         cin >> a[i][j];
   sort2D(a);
   return 0;
}
```

Defining and Calling Functions

Correct

```
void f() {
}

int main() {
  f();
  return 0;
}
```

Syntax Error

```
int main() {
  f();  // f() is undefined
  return 0;
}

void f() {
}
```

A function should be **defined before use**.

Function Prototype

- C++ language allows us to define the function prototype without implementation, and then call the function
- Function prototype
  - Specifies the function name, input and output type only.
  - The following statement specifies that **f** is a function, there is no input and no return value

```
void f(void);
```

- The function can be implemented later

Function Prototype

```
void f (void);

int main() {
  f();
  return 0;
}

void f() {
  //define f() here
}
```

```
int findMax (int, int);

int main() {
  int x = findMax (3, 4);
  return 0;
}

int findMax (int n1, int n2) {
  //define findMax() here
}
```

Function Prototype

- The prototype

```
int findMax(int, int);
```

  - specifies that **`findMax`** is a function name
  - return type is **`int`**
  - there are ***two arguments*** and their types are **`int`**.

- Another way to write the prototype is:

```
int  findMax(int n1, int n2);
```

  - The variable names are optional.

Function Prototype

- In C++, function prototypes and definitions can be stored separately
- Header files (**`.h`**):
  - With extension **`.h`**, .e.g `mylib.h`, `myclass.h`
  - Contain function prototypes only
  - To be included in the program that will call the function
- Implementation file (**`.cpp`**):
  - Contain function implementation (definition)
- The name of **`.h`** and **`.cpp`** files should be the same

Function Prototype

main.cpp

```
#include "mylib.h"
int main() {
   int x, y = 2, z = 3;
   …
   x = calMin(y, z);
   …
   return 0;
}
```

mylib.h

```
int calMin(int, int);
```

mylib.cpp

```
int calMin(int a, int b) {
   if (a > b)
      return b;
   else
      return a;
}
```

Function prototype (cont'd)

- `void` is used if a function has no return value
- Prototypes allow the compiler to check the code more thoroughly
- Values passed to a function are coerced where necessary, e.g., `printDouble(4)` where the integer `4` will be promoted as a double data type `4`

```
#include <iostream>
using namespace std;
void printDouble(double d){
    cout << fixed;
    cout << d <<endl;
}
int main() {
  int x = 4;
  printDouble(x);
  return 0;
}
```

Recursions

- One basic problem solving technique is to break the task into subtasks
- If a subtask is a smaller version of the original task, you can solve the original task using a recursive function
- A recursive function is one that invokes itself, either directly or indirectly

Example: Factorial

- The factorial of n is defined as:

  0! = 1

  n! = n*(n-1)*…2*1 for n > 0

- A recurrence relation: (induction)
  - **n! = n*(n-1)! for n > 0**
  - E.g.:

  3! = 3 * 2!

  = 3 * 2 * 1!

  = 3 * 2 * 1 * 0!

  = 3 * 2 * 1 * 1

Iterative vs. Recursive

Iterative

```
int factorial(int n) {
   int i, fact = 1;
   for (i = 1; i <= n; i++) {
     fact = i * fact;
   }
   return fact;
}
```

Recursive

```
int factorial(int n) {
   if (n == 0)
     return 1;
   return n * factorial(n-1);
}
```

Checkpoints

1. There is no infinite recursion (check exit condition)
2. Each stopping case performs the correct action for that case
3. For each of cases that involve recursion, if all recursive calls perform their actions correctly, then the entire case performs correctly.

Summary

- Functions help programmer write a more simple program and make the problem easier to solve
- `return_type function_name(paramaters);`
- Function prototype must declared before it can be used.
- Header files can be used to store function prototypes but not the body.
- Parameters can be passed with call by value or call by reference.