CS2310 Computer Programming

LT05: Function

Computer Science, City University of Hong Kong (Dongguan)

Semester A 2025-26

Outline

- Function declaration

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

- Recursive function

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
  - 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

Function in C++

- C++ standard library provides a rich collection of functions
- Mathematical calculations (#include <cmath>)
- String manipulations (#include <cstring>)
- Input/output (#include <iostream>)

How to use function written by others?

```
#include <iostream>
using namespace std;
void main()
{
   float area,side;
   cout << "Enter the area of a square:";
   cin >> area;
   side=sqrt(area);
   cout << "The square has perimeter: " << 4*side;
}
```

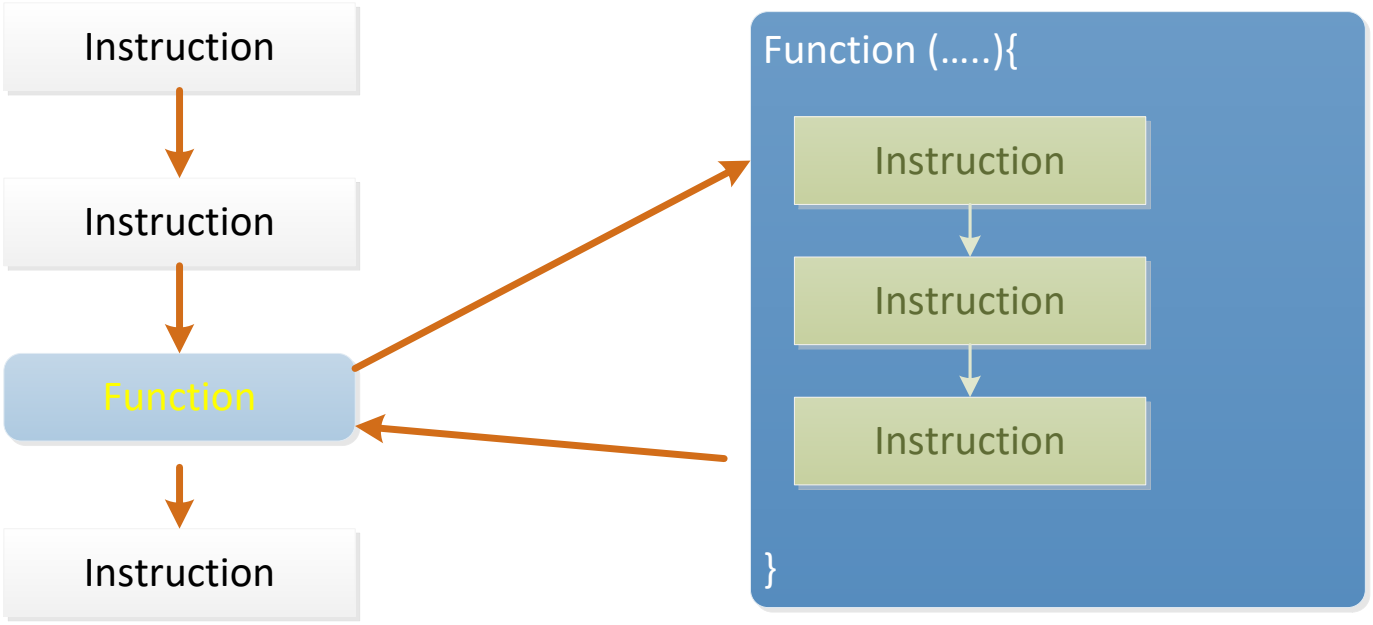
Tell compiler that you are going to use functions defined in iostream package

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

Code: lec05-01-sqrt.cpp

Function invocation

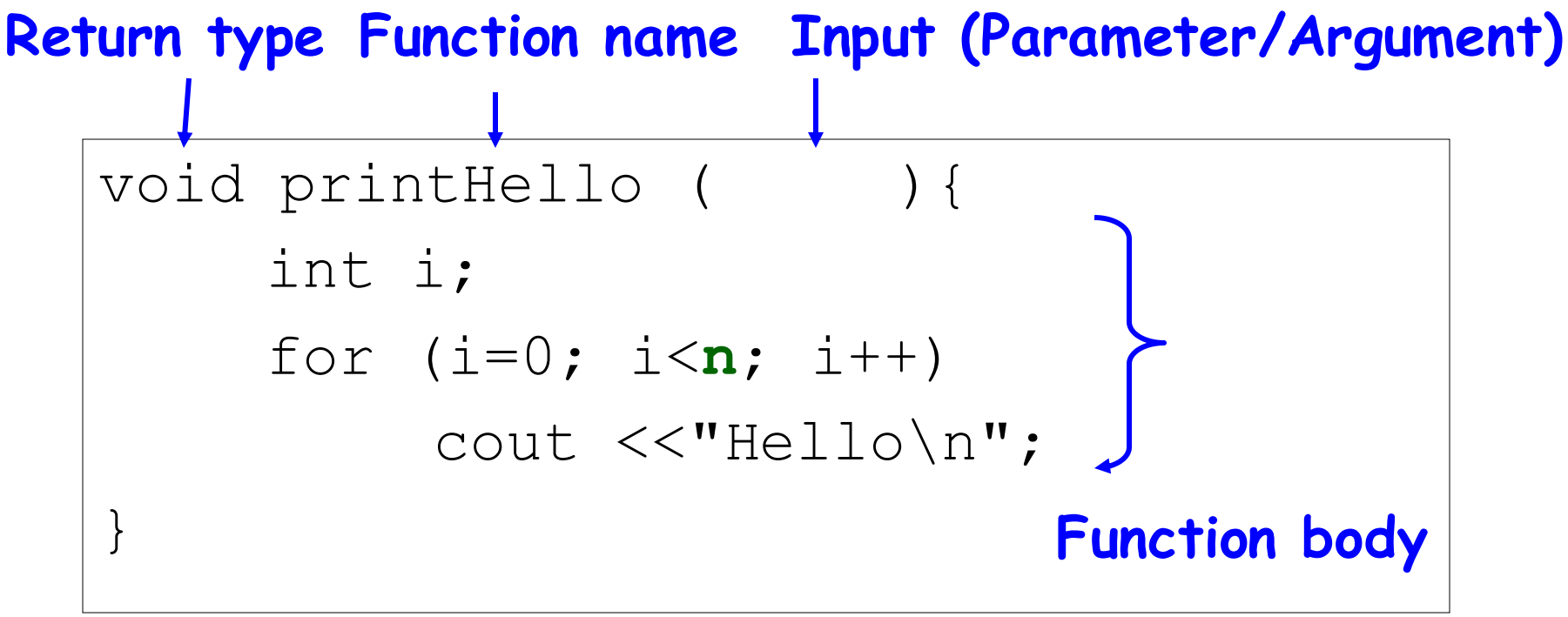
- During program execution, when a function name followed by parentheses is encountered, the function is invoked and the program control is passed to that function; when the function ends, program control is returned to the statement immediately after the function call in the original function



Write your own function (User defined)

- Define a function `printHello`, which accepts an integer `n` as input
- The function should print `"Hello"` `n` times, where `n` is an integer

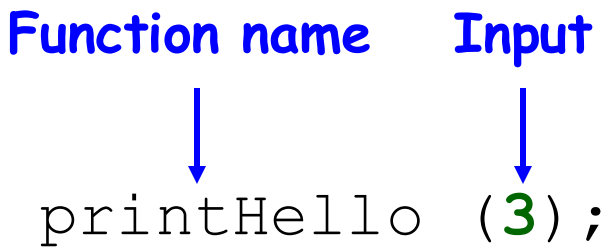
Function Components



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

Calling a function (I)

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



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

Syntax error:

```
void printHello (3);
```

Calling a function (II)

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

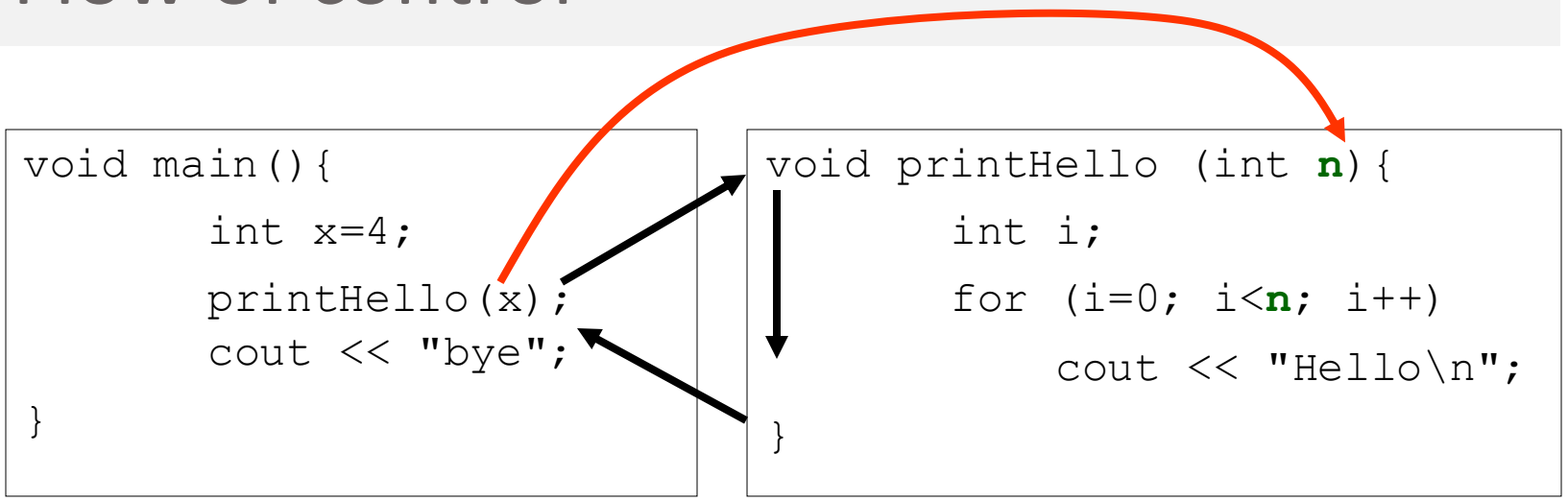
Print "hello" 4 times and then 6 times.
We **don't** need the parameter type when calling a function.

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

Outline

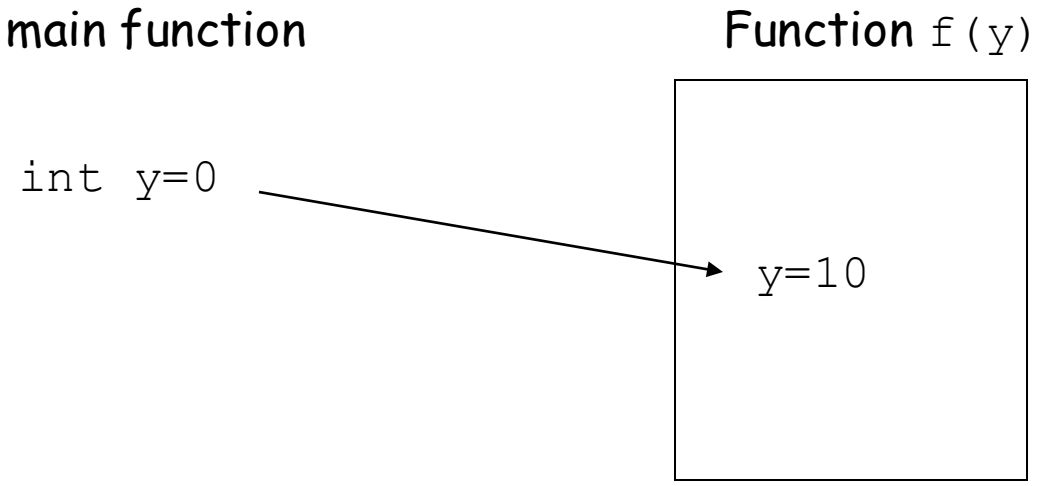
- Function declaration

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

- Recursive function

Parameters Passing: Call-by-Value

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



Code: lec05-02-value.cpp

Local to the function

```
void f (int y){
  y=10; //modify y in f(), not the one in main()
}

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

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 `main()`?

- 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, if necessary, to the type specified in the function definition
- Syntax:

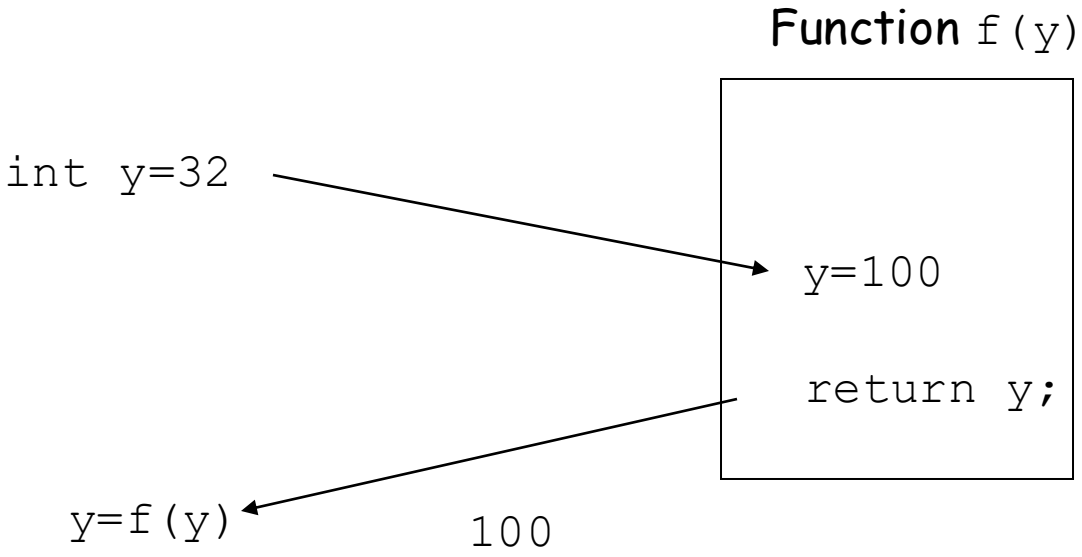
```
return expression;
return;
```

- Example:

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


Code: lec05-03-return.cpp

Return value



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

How to modify y in `main()`?

```
int f (int x){
  x=4; //we modify the value x to 4

  return x;
}

void main(){
  int y=3;
  y=f(y);
  cout << y;
}
```

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

Function definition

```
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);
```


Code: lec05-04-findmax.cpp

Parameters Passing: Call-by-Pointer

- Add the '*' sign to the function parameters that store the variable call by pointer

```
void cal(int *x, int y){
    //x is call by pointer
    //y is call by value
    ……
    ……
}
```

- Add the '&' sign to the variable when it needs to be called by pointer

```
int x=0, y=1;
cal(&x, y);
```


Code: lec05-05-pointer.cpp

Call by Pointer

```
void f (char *c1_ptr){
   *c1_ptr='B';
}

void main(){
   char c1_in_main='A'; // c1_in_main =65
   f(&c1_in_main);
   cout << c1_in_main;
}
```

When f() is called, the following operation is performed

c1_ptr = **&c1_in_main**;

Call by Pointer

```
void f (char *c1_ptr){
   *c1_ptr='B';
}

void main(){
   char c1_in_main='A'; // c1_in_main =65
   f(&c1_in_main);
   cout << c1_in_main; //print 'B'
}
```

points to

Variable	Variable type	Memory location	Content
c1_in_main	char	**3A8E**	65
c1_ptr	char pointer	4000	

Assign location 3A8E to c1_ptr

Location of c1_in_main (location 3A8E) is assigned to c1_ptr1

c1_ptr = **&c1_in_main**;

Call by Pointer

```
void f (char *c1_ptr){
   *c1_ptr='B';
}

void main(){
   char c1_in_main='A'; // c1_in_main =6
   f(&c1_in_main);
   cout << c1_in_main; //print 'B'
}
```

update

c1_ptr points to location 3A8E (that is the variable c1_in_main).
*c1_ptr refers to the variable pointed by c1_ptr, i.e. the variable stored at 3A8E

Variable	Variable type	Memory location	Content
c1_in_main	char	3A8E	66
c1_ptr	char pointer	4000	3A8E

```
c1_ptr='B'; //error
```

Reason: c1_ptr1 stores a **location** so it cannot store a **char** (or the ASCII code of a char)

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

Code: lec05-06-default.cpp

Parameter Passing: Default Parameters

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

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

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
  }
```

Outline

- Function declaration

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

- Recursive function

Parameters Passing: arrays

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

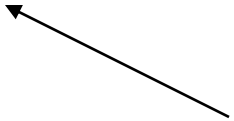
```
void f(int x[]){
  …
}

void main(){
  int y[20];
  f(y[0]);
}
```

Parameters Passing: arrays

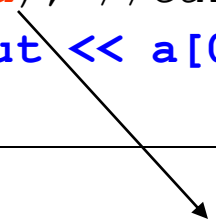
The size of array is optional

```
void f(int a[])
```



```
void f(int a[3]){
   cout << a[0] <<endl; //1 is printed
   a[0]=10;
}

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



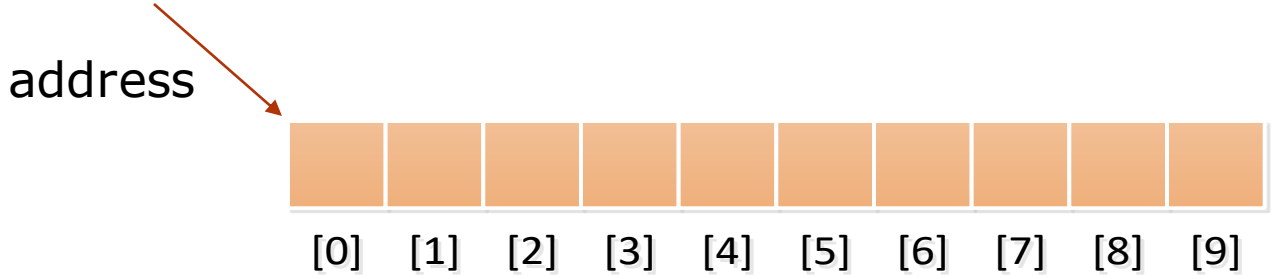
Only need to input the array name!

Code: lec05-07-array.cpp

Parameters Passing: arrays

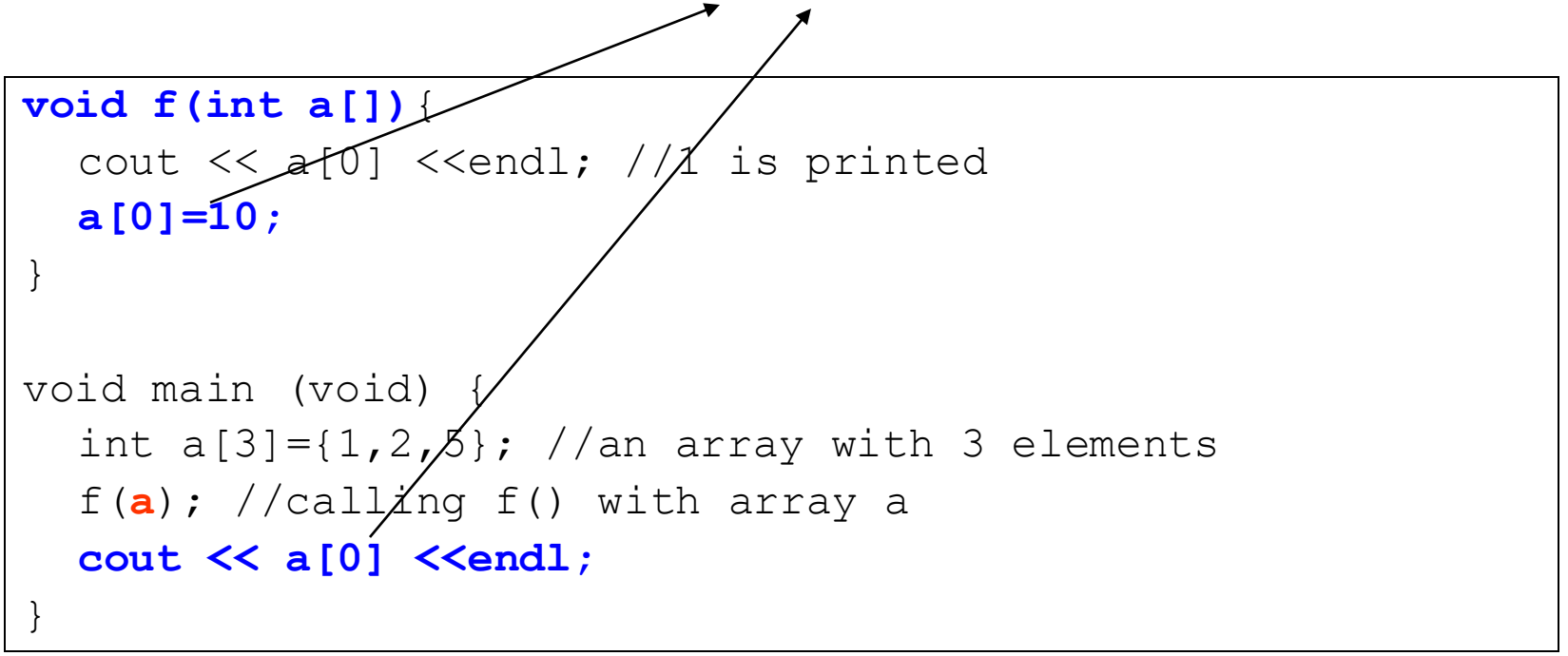
- Name of an array
  - `char name[] = "CityU";` or
  - `char * name = "CityU";`

  - **name** ???



Parameters Passing: arrays

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



```
void f(int a[]){
   cout << a[0] <<endl; //1 is printed
   a[0]=10;
}

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

Parameter Passing: 2D arrays

- The way to pass a 2D array is similar as the 1D array
- For 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[][4]){
  …
  }

void main(){
  int y[3][4] = {{6, 2, 9, 3},{2, 8, 1, 0},{5, 3,
  7, 8}};
  sort2D(y);
}
```

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

Code: lec05-08-2D.cpp

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;
        }
}
void 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);
}
```

Defining and calling functions

Correct

```
void f(){
}

void main(){
        f();
}
```

Syntax Error

```
void main(){
        f(); //f() is undefined
}

void f(){
}
```

A function should be defined before use

Function Prototype

- C++ language allows us to define the function prototype and then call the function
- Function **prototype**
  - Specifies the function name, input and output type
  - The following statement specifies that f is a function, there is no input and no return value

```
void f();
```

- The function can be implemented later

Function Prototype

```
void f ();

void main(){
        f();
}

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

```
int findMax (int, int);

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

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

Function Prototype

The prototype

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

specifies that `findMax` is a function name

The 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);
```

However, the variable names are optional

Function Prototype

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

Code: lec05-09-prototype.cpp

Function Prototype

main.cpp

```
#include "mylib.h"
void main(){
   int x,y=2,z=3;

   …….
   x=calMin(y,z);
   …….

}
```

mylib.h

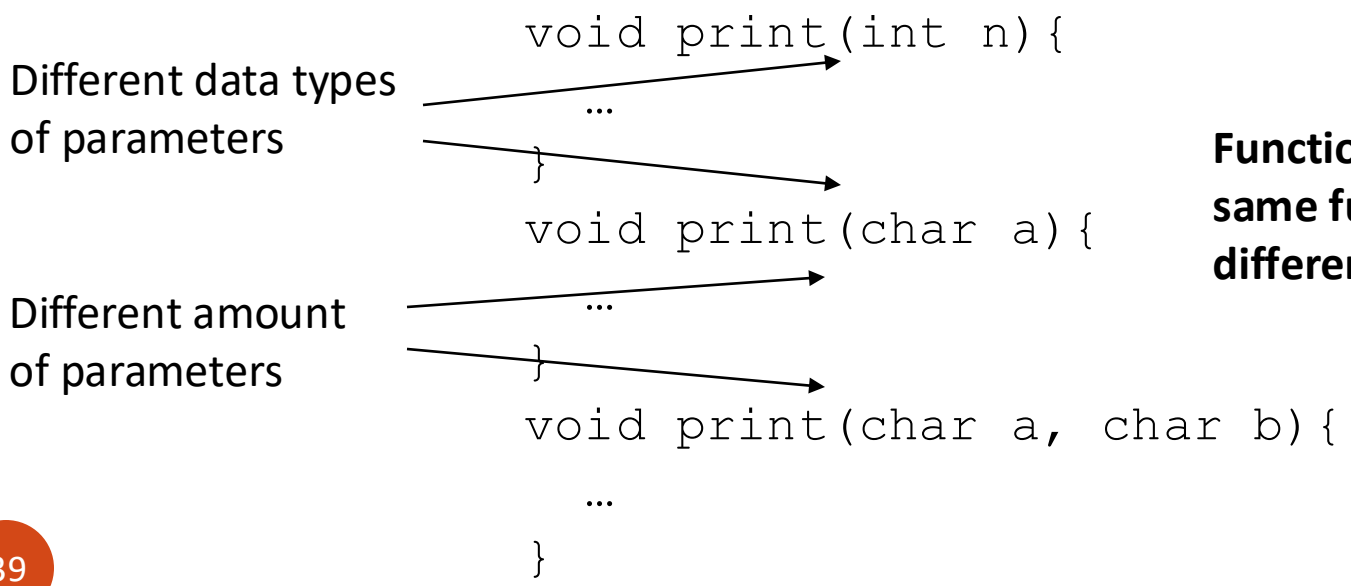
```
int calMin(int,int);
```

mylib.cpp

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

Function overloading

- For functions with similar purposes, it will be more convenient to use the same function name
- C++ allows two different functions share the same function name while their parameters are different (amount or data type of the parameter), which we call as **function overloading**



Function overloading: the same function name, but different functions

Function overloading

- It is invalid that two functions have the same function names and parameters but different return types

```
void print(int n){
  …
  }

int print(int n){
   …
  }
```

Compilation error: the function name and the parameter are both the same

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

Outline

- Function declaration

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

- Recursive function

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
- A base case should eventually be reached, at which time the breaking down (recursion) will stop

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
```


Code: lec05-10-recursion.cpp

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);
}
```

Example

- Write a function int getValue(int *num*)
  - *num is **n**-digit integer, where **n**>=1*
  - **Calculate** the value of the last half digits of *num*
- Last half digits
  - If ***n*** is odd, it is "***n***/2 + 1", e.g., *num* = 51176 and ***n*** = 5
    - Last half digits are: 176
  - If ***n*** is even, it is "***n***/2", e.g., *num* = 6734 and ***n*** = 4
    - Last half digits are: 34
  - If *num* = 10030 and ***n*** = 5, output *30* (**not** *030*)
- **Key idea: *num* % 10^(half length of *n*)**

Code: lec05-11-value.cpp

Checkpoints

1. There is no infinite recursion (check exist 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

Code: lec05-12-func.cpp

Example: Exponential func.

- How to write `exp(int n, int p)` recursively?
- E.g., exp(*n*, *p*) = n^p

$$n^p = n \times n^{p-1} \qquad \exp(n, p) = n \times \exp(n, p-1)$$

```
int exp(int n, int p){
    if(p == 0)
        return 1;
    return n * exp(n, p-1);
}
```


Code: lec05-13-exp.cpp

Example: number of zero

- Write a recursive function that **counts** the number of zero digits in an integer, *n*
- E.g., if *n* is 10200, **`zeros(10200)`** returns 3
- How to **decompose**?
  - Last digit + rest digits
  - E.g., 10200 → 1020 & 0
- Base cases?
  - If digit is ?, return ?
  - If digit is ?, return ?

```
zeros(10200)
zeros(1020)                                  + zeros(0)
zeros(102)                          + zeros(0) + zeros(0)
zeros(10)              + zeros(2) + zeros(0) + zeros(0)
zeros(1) + zeros(0) + zeros(2) + zeros(0) + zeros(0)
```


Code: lec05-14-zero.cpp

Example: number of zero

- Write a recursive function that **counts** the number of zero digits in an integer
- **`zeros(10200)`** returns 3

```
int zeros(int numb){
   if(numb==0)            // 1 digit (zero/non-zero):
      return 1;           // bottom out.
   else if(numb < 10 && numb > -10)
      return 0;
   else                   // > 1 digits: recursion
      return zeros(numb/10) + zeros(numb%10);
}
```

Example: printing

- Printing a message for *n* times
  - E.g., `void nPrintln(char* message, int n)`

- We can break the problem into two sub-problems:
  - One is to print the message one time and
  - The other is to print the message for *n-1* times.
  - The second problem is the same as the original problem with a smaller size. The base case for the problem is n==0

```
void nPrintln(char * message, int times)
{
  if (times >= 1) {
    cout << message << endl;
    nPrintln(message, times - 1);
  } // The base case is n == 0
}
```

Code: lec05-15-print.cpp

Static variable in function

- Question: In C++, all the variable inside a function call will be deleted by the compiler after the execution of the function block. How to count how many function calls when running a recursion?
- The keyword `static` will keep the variable after it exists even after the function call

```
int factorial(int n){
  static int count;
  count++;
  …
}
```

Compiler will assign 0 to static variable if no initialization.

Static variable which will not disappear after the execution of the current function call. Hence, it can be used to count the number of function calls during a recursion

Code: lec05-16-count.cpp