

# Introduction to Object-Oriented Programming

## COMP2011: C++ Function III — Declaration, Definition, Overloading, and Default Arguments

Dr. Cindy Li  
Dr. Brian Mak  
Dr. Dit-Yan Yeung

Department of Computer Science & Engineering  
The Hong Kong University of Science and Technology  
Hong Kong SAR, China



# Part I

## Function Declaration and Function Definition

# Some Function Terminology

function prototype

`int max(int, int);`

↑  
return  
type

↑  
name

signature

A handwritten diagram illustrating the components of a function prototype. The text 'function prototype' is written in pink at the top, with a pink bracket underneath it spanning the width of the code below. The code 'int max(int, int);' is written in blue. Below the code, three annotations are present: an orange arrow points from the word 'return' (part of 'return type') to the 'int' at the beginning of the code; another orange arrow points from the word 'name' to the 'max' in the code; and a green bracket is placed under the parameters '(int, int)', with the word 'signature' written in green below it.

# Function Prototype

A **function prototype** consists of

- 1 **function name**
- 2 **return data type**
- 3 the **number** of formal parameters
- 4 the **data type** of the formal parameters

## Example: Function Prototypes

```
// int factorial(int n) { ... }
```

```
int factorial(int);
```

```
// float distance(float x1, float y1, float x2, float y2) { ... }
```

```
float distance(float, float, float, float);
```

```
/* void print_tree(int tree_height, char tree_symbol,  
    char trunk_symbol, char pot_symbol) { ... } */
```

```
void print_tree(int, char, char, char);
```

# Function Prototype ..

- The **identifier names** of the formal parameters are not part of the signature as the names are **immaterial**.

## Example: Variable Names are Immaterial in a Function Prototype

*/\* All the following 3 function definitions are equivalent \*/*

```
int max(int x, int y) { return (x > y) ? x : y; }
```

```
int max(int a, int b) { return (a > b) ? a : b; }
```

```
int max(int f, int g) { return (f > g) ? f : g; }
```

- A **function prototype** describes the **interface** of the function: what parameters it takes in and what value it returns.
- Technically, a **function prototype** is also called the **application programming interface** (API).

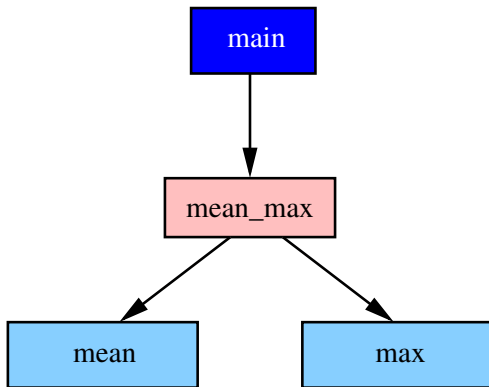
# Function Declaration vs. Definition

- A function is **declared** by writing down its interface — its **function prototype**.
- A function is **defined** by writing down its **function header** *plus* its **function body**.
- A **function definition** will ask the compiler to generate **machine codes** according to the C++ codes in its function body.
- A **function declaration** just informs the compiler about the function's **interface** *without* generating any machine codes.
- A function may be **declared many times**, but a function can be **defined only once**.
- Of course, when a function is **defined**, it is also **declared**.
- But, simply **declaring** a function does **not define** the function.

# Function Declaration vs. Definition ..

- In C++, all functions must be **declared before** it can be used, so that the compiler can
  - make sure the exact **number of arguments** are passed.
  - do **type checking** on the arguments passed to the function.
- That is, if function A wants to call function B, function B must be
  - **declared/defined before**, or
  - **declared inside** function A **before** calling function B.
- However, a function need not be defined before it can be used, although it must be defined **eventually somewhere** in the whole program in order that the program can be compiled to an executable.

## Example: A Program with 3 Levels of Functions





# Example: Declare Functions by Defining the Functions

```
#include <iostream>                                /* File: fcn-prototype1.cpp */
using namespace std;

int max(int x, int y) { return (x > y) ? x : y; }
int mean(int x, int y) { return (x + y)/2; }

void mean_max(int x, int y, int& mean_num, int& max_num)
{
    mean_num = mean(x, y);
    max_num = max(x, y);
}

int main(void)
{
    int average, bigger;

    mean_max(6, 4, average, bigger);
    cout << "mean = " << average << endl << "max = " << bigger << endl;
    return 0;
}
```

# Example: Declare Functions Globally

```
#include <iostream>                                     /* File: fcn-prototype2.cpp */
using namespace std;

void mean_max(int, int, int&, int&);    // main only needs to know mean_max

int main(void)
{
    int average, bigger;
    mean_max(6, 4, average, bigger);
    cout << "mean = " << average << endl << "max = " << bigger << endl;
    return 0;
}

int max(int, int);                                     // mean_max needs to know max and mean
int mean(int, int);

void mean_max(int x, int y, int& mean_num, int& max_num)
{
    mean_num = mean(x, y);
    max_num = max(x, y);
}

int max(int x, int y) { return (x > y) ? x : y; }
int mean(int x, int y) { return (x + y)/2; }
```

# Example: Declare Functions Locally

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

```
/* File: fcn-prototype3.cpp */
```

```
int main(void)
{
    void mean_max(int, int, int&, int&);
    int average, bigger;

    mean_max(6, 4, average, bigger);
    cout << "mean = " << average << endl << "max = " << bigger << endl;
    return 0;
}
```

```
void mean_max(int x, int y, int& mean_num, int& max_num)
{
    int max(int, int);
    int mean(int, int);

    mean_num = mean(x, y);
    max_num = max(x, y);
}
```

```
int max(int x, int y) { return (x > y) ? x : y; }
int mean(int x, int y) { return (x + y)/2; }
```

## Example: Forward Function Declaration

```
#include <iostream>                                /* File: odd-even.cpp */
using namespace std;

bool even(int);

bool odd(int x) { return (x == 0) ? false : even(x-1); }

bool even(int x) { return (x == 0) ? true : odd(x-1); }

int main(void)
{
    int x;
    cin >> x;                                       // assume x > 0
    cout << boolalpha << odd(x) << endl;
    cout << boolalpha << even(x) << endl;
    return 0;
}
```

## Part II

# Function Overloading



# Signature of a Function

- Recall that in C++, all functions are **global**. That means, in general, all functions can “see” each other.
- Just as we use one’s signature to identify the person, we identify a function by its **name** and **signature**.
- A function’s **signature** is the list of **formal parameters** without their identifier names.
- **No** two C++ functions can have the **same name** *and* **same signature** but **different return type**.
- **BUT** two C++ functions can have the **same name** *but* **different signature**  $\Rightarrow$  **function overloading**.

## Example: No 2 Function Prototypes Differ Only in Return Type

```
/* the following 2 function definitions of pick_one cannot appear  
   in the same program */  
int pick_one(int x, float y) { return x; }  
float pick_one(int x, float y) { return y; }
```

# Function Overloading

C++ allows **several functions** to have the **same name** but **different types** of input parameters.

## Example: Overloaded Functions

```
int max(int x, int y) { return (x > y) ? x : y; }
int max(int x, int y, int z) { return max(max(x,y), z); }
double max(double a, double b) { return (a > b) ? a : b; }

void swap(int& a, int& b) { int temp = a; a = b; b = temp; }
void swap(float& a, float& b) { float temp = a; a = b; b = temp; }
void swap(double& a, double& b) { double temp = a; a = b; b = temp; }

int absolute(int a) { return (a < 0) ? -a : a; }
int absolute(int& a) { return (a = (a < 0) ? -a : a); }
```

**Question:** How can you call the version

`int absolute(int&);`

# Example: Invalid Function Overloading

*/\* identifier names of formal parameters are immaterial \*/*

```
int max(int x, int y) { return (x > y) ? x : y; }
```

```
int max(int a, int b) { return (a > b) ? a : b; }
```

*/\* return type is not part of the signature \*/*

```
void swap(int& a, int& b) { int temp = a; a = b; b = temp; }
```

```
int swap(int& a, int& b) { int temp = a; a = b; b = temp; return a; }
```



# Overloaded Function Resolution

- When an **overloaded function** is called, C++ will determine exactly which function among those with the same name should be called — **function resolution**.
- **Function resolution** is done by analyzing the type of actual parameters being passed during a function call.
- If there are **exact matches** between the types of formal parameters in an **overloaded function** and those of actual parameters, that function is picked.
- If there is no exact match, C++ uses a set of **pre-defined rules** for **function resolution**.
- The basic idea is to try to match their data types by:
  - first, **widening conversion** (coercion)
  - then, **narrowing conversion** (coercion)

# Example: Function Resolution

```
int test(int a, double b);  
int test(double a, int b);
```

- If you make the following function call: `test(3, 4.6)`, the compiler will pick the **first version**.
- If you make the following function call: `test('a', 4.6)`, the compiler will again pick the **first version** by converting 'a' to an int.
- If you make the following function call: `test(3.2, 4.6)`, it can either
  - match to the first version by narrowing conversion of the first parameter to int.
  - match to the second version by narrowing conversion of the second parameter to int.
  - since neither one is more preferable than the other one  
⇒ **compilation error!**

# Default Function Argument

- Sometimes, we would like a function to have certain **default** behaviour, but still allow the user to **change** it.
- C++ allows the user to call a function with **fewer arguments** if all he wants is its **default behaviour**, and with **more arguments** if he wants some **particular behaviour** of the function.
- A function may have more than 1 **default argument**.
- But all **default arguments** must be specified at the **end** of the **formal parameter** list.

```
/* The following 2 prototypes are equivalent */  
void func(int x, float& y, char gender = 'M', bool alive = true);  
void func(int, float&, char = 'M', bool = true);
```

- The default argument(s) may be specified in a **function declaration** or **function definition**, but not both.
  - usually we put it on the **function declaration**. **Why?**
- A function with **default arguments** looks like several **overloaded functions**, but it is not.

## Example: getline( ) Again

- The true `getline` function header is:

```
cin.getline(char s[], int max-num-char, char terminator='\n');
```

- Thus, you may call it as

```
cin.getline(char s[], int max-num-char);
```

and the default terminator is the newline character.

### Example: Different getline Calls

```
const int MAX_LINE_LEN = 1000;  
char s[MAX_LINE_LEN+1];
```

```
cin.getline(s, MAX_LINE_LEN+1);  
cin.getline(s, MAX_LINE_LEN+1, '.');
```

```
// terminator = newline  
// terminator = full stop
```

## Example: Increment with Default Argument

```
#include <iostream>      /* File: increment-default-arg.cpp */
using namespace std;

int increment(int x, int step = 1)
{
    return (x + step);
}

int main(void)
{
    cout << increment(10) << endl;
    cout << increment(10, 5) << endl;

    return 0;
}
```

# Example: Sort with Default Argument

```
#include <iostream>                                     /* File: sort-default-arg.cpp */
using namespace std;

enum sort_order {ASCENDING, DESCENDING};

void swap(int& a, int& b) { int temp = a; a = b; b = temp; }

void sort(int& x, int& y, sort_order order = ASCENDING)
{
    if (order == ASCENDING) {
        if (x > y) swap(x, y);
    }
    else {
        if (x < y) swap(x, y);
    }
}

int main(void)
{
    int a = 24, b = 8;
    sort(a, b); cout << a << '\t' << b << endl;
    sort(a, b, DESCENDING); cout << a << '\t' << b << endl;
    return 0;
}
```