Functions Overview

(CS 1002)

Dr. Muhammad Aleem,

Department of Computer Science,

National University of Computer & Emerging Sciences,
Islamabad Campus

Functions in C++

- It is better to **develop** and **maintain large programs** in the form of **smaller pieces** (**modules**)
- This technique Called "**Divide and Conquer**"

A Development Approach

```
main()
{
    -----
    -----
    -----
    -----
    .
    .
    .
    ----
    -----
    -----
    return 0;
}
```

Easier To >>

- ✓Design
- ✓Debug
- ✓Extend
- ✓Modify
- ✓Understand
- ✓Reuse

Better Development Approach

```
main()
{
   -----
    ----
}
```

```
function f1()
{
    ---
    ---
}
```

```
function f2()
{
    ---
    ---
}
```

Functions in C++(Cont.)

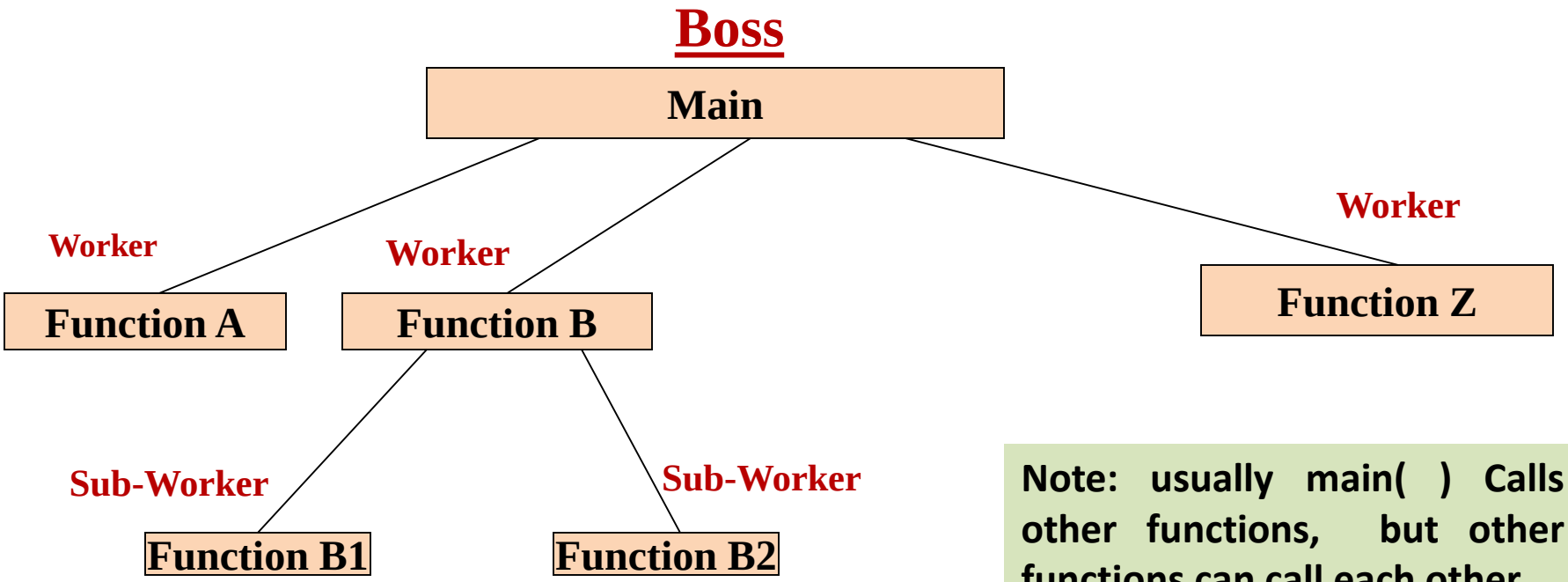
- In C++ **modules** Known as **Functions** & **Classes**

- Programs may use **new** and "**prepackaged**" or built-in **modules**
  - **New**: **programmer-defined functions** and **classes**
  - **Prepackaged**: from the ***standard library***

About Functions in C++

- **Functions invoked** by a **function–call-statement** which **consist** of it's **name** and **information** it **needs** (*arguments)*

Example:

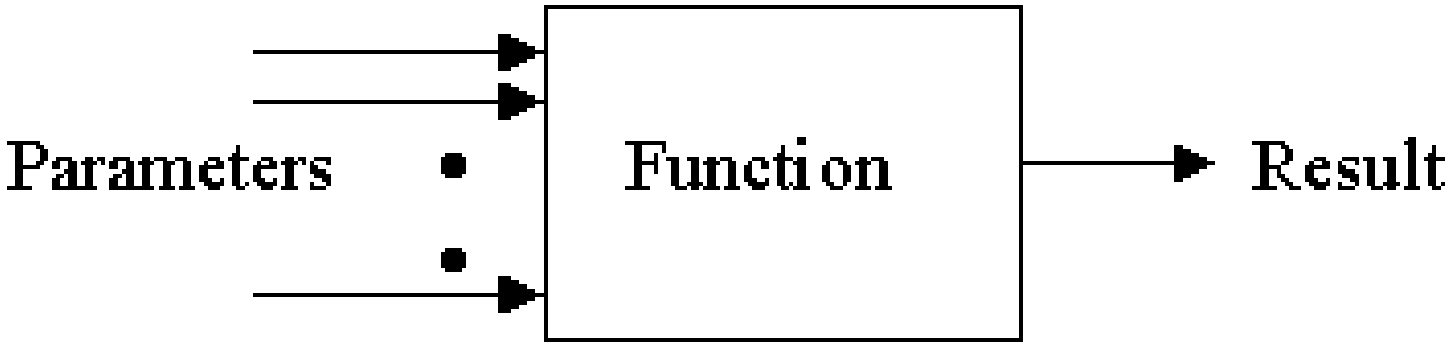


Note: usually main( ) Calls other functions, but other functions can call each other

Calling Function

Function calls:

Provide **function name** and **arguments (data); Function performs operations** and; Function ***returns results***



Calling Functions

- **Functions calling (Syntax):**

<function name> (<argument list>);

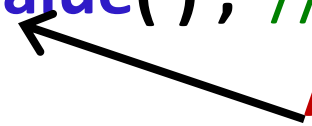
e.g.,

```
FunctionName( );
FunctionName(argument1);
FunctionName(argument1, argument2, …);
. . .
```

Calling Functions

- **Examples (built-in, and user-defined functions)**

```
int n = getPIValue( ) ;  //takes no argument
```



A user-defined function

```
cout << sqrt(9);  //takes one argument, returns square-root

cout<<pow(2,3); //calculates 2 power 3

cout<<SumValues(myArray); //returns sum of the array
```

A user-defined function

Function Definition

Syntax for function definition:

```
returned-value-type function-name (parameter-list)
{
    Declarations of local variables and Statements;
    …
}
```

- **Parameter list**
  - **Comma separated** list of **arguments**
    - **Data type** needed for **each argument**
  - If **no arguments** → **leave blank**

- **Return-value-type**
  - **Data type** of **result returned** (use **void** if nothing will be returned)

Function Prototype

- Before a **function** is **called**, **it must be declared first**.
- **Functions cannot** be **defined inside other functions**
- A **function prototype** is a ***function declaration without implementation***:

```
int multiplyTwoNums(int,int);
```

A Function prototype (declaration without implementation)

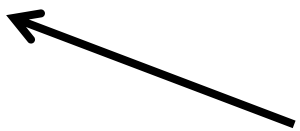
Function Prototype (cont.)

- **Why it is needed?**

  It is required to declare a function prototype before the function is called.

Function Prototype (cont.)

```
int main() {
   int sum = AddTwoNumbers(3,5);
   cout<<sum;
   return 0;
}


int AddTwoNumbers(int a, int b) {
      int sum = a+b;
      return sum;
}
```



Error: Un-defined function

Function Prototype (cont.)

Solution-1

```
int AddTwoNumbers(int a, int b) {
    int sum = a+b;
    return sum;
}


int main() {
    int sum = AddTwoNumbers(3,5);
    cout<<sum;
    return 0;
}
```

Solution-2

```
int AddTwoNumbers(int, int);

int main() {
    int sum = AddTwoNumbers(3,5);
    cout<<sum;
    return 0;
}


int AddTwoNumbers(int a, int b) {
    int sum = a+b;
    return sum;
}
```

Function signature and Parameters

- **Function signature** is the **combination** of the **function name** and the **parameter list**.

- **Variables** defined in the **function header** are known as **formal parameters.**

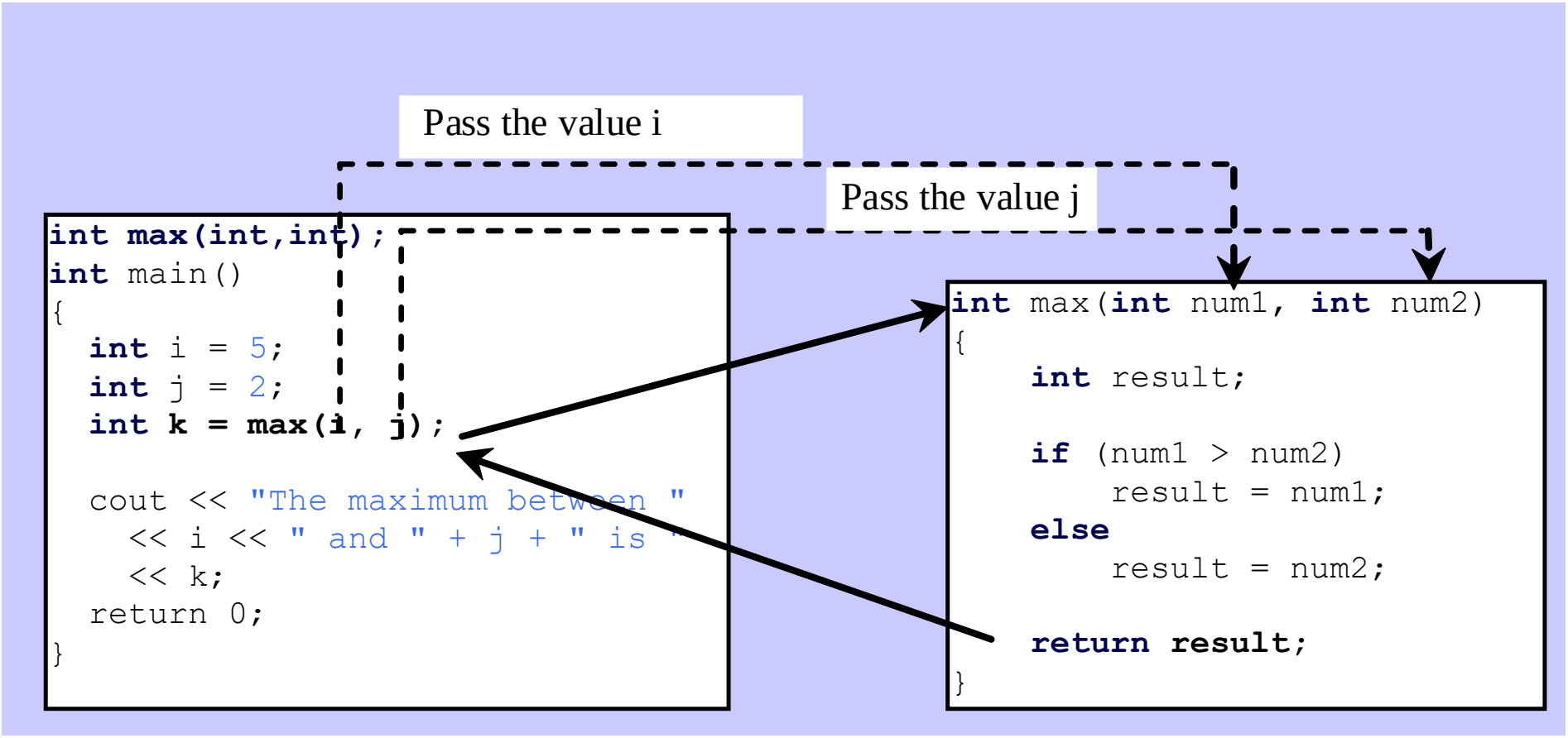
- When a **function** is **invoked**, you **pass a value** to the parameter. This **value** is **referred** to as **actual parameter or argument**.

Function's return values

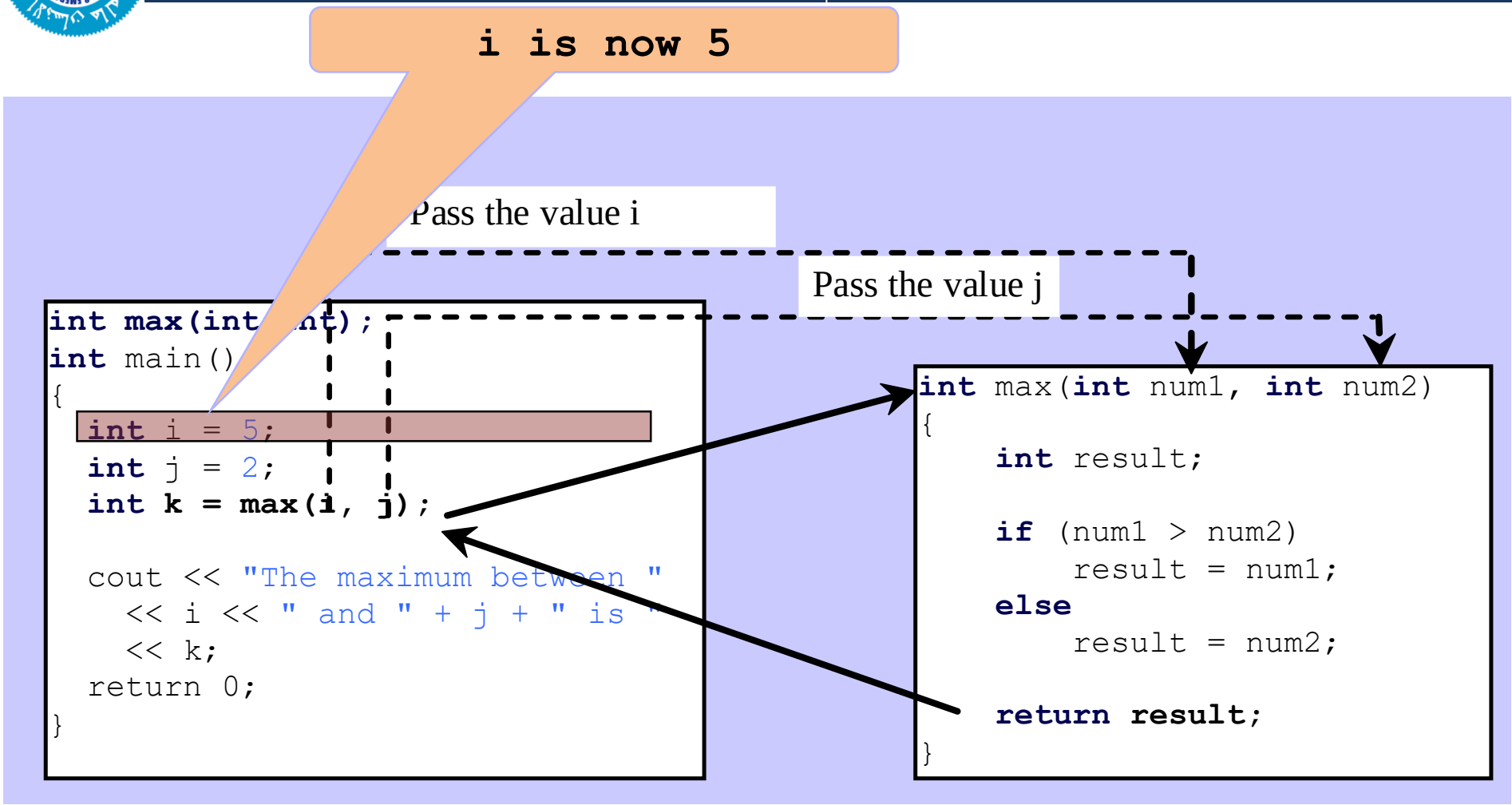
- A **function** may **return** a **value**:
  - ***returnValueType*** is the **data type** of the **value** the **function returns**.

- If **function** does **not return a value**, the ***returnValueType*** is the keyword **void**.

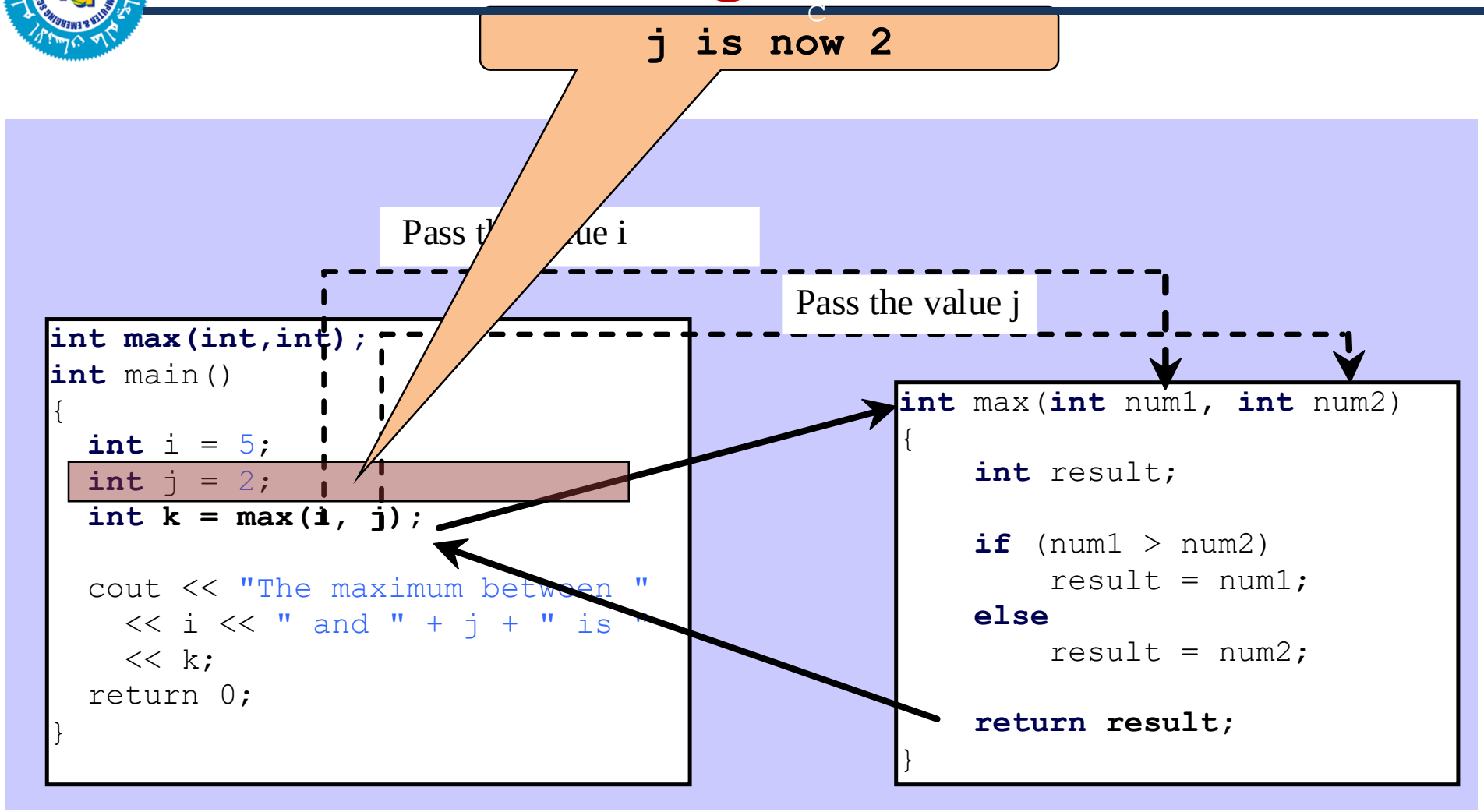
Calling Functions



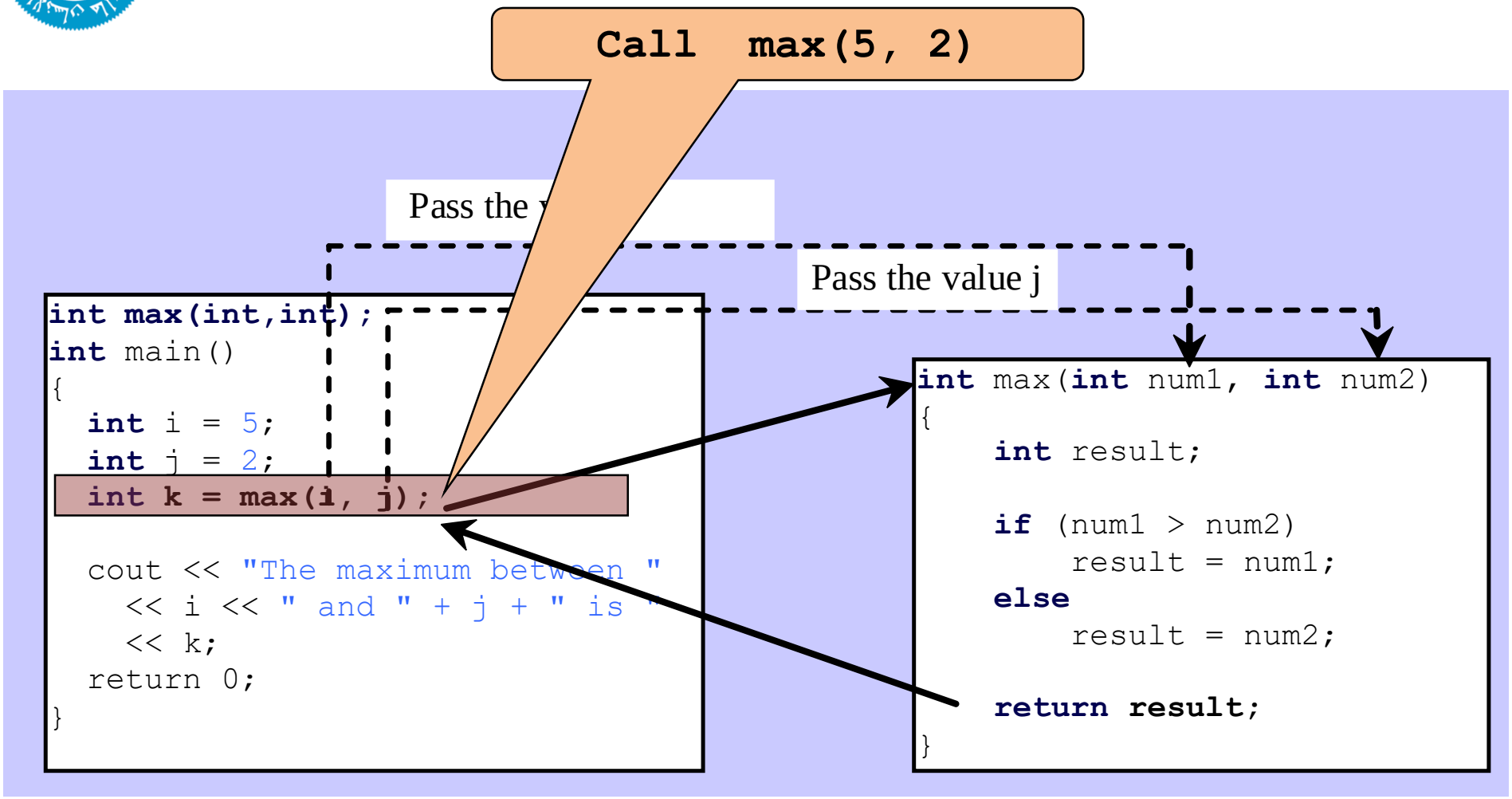
Calling Functions



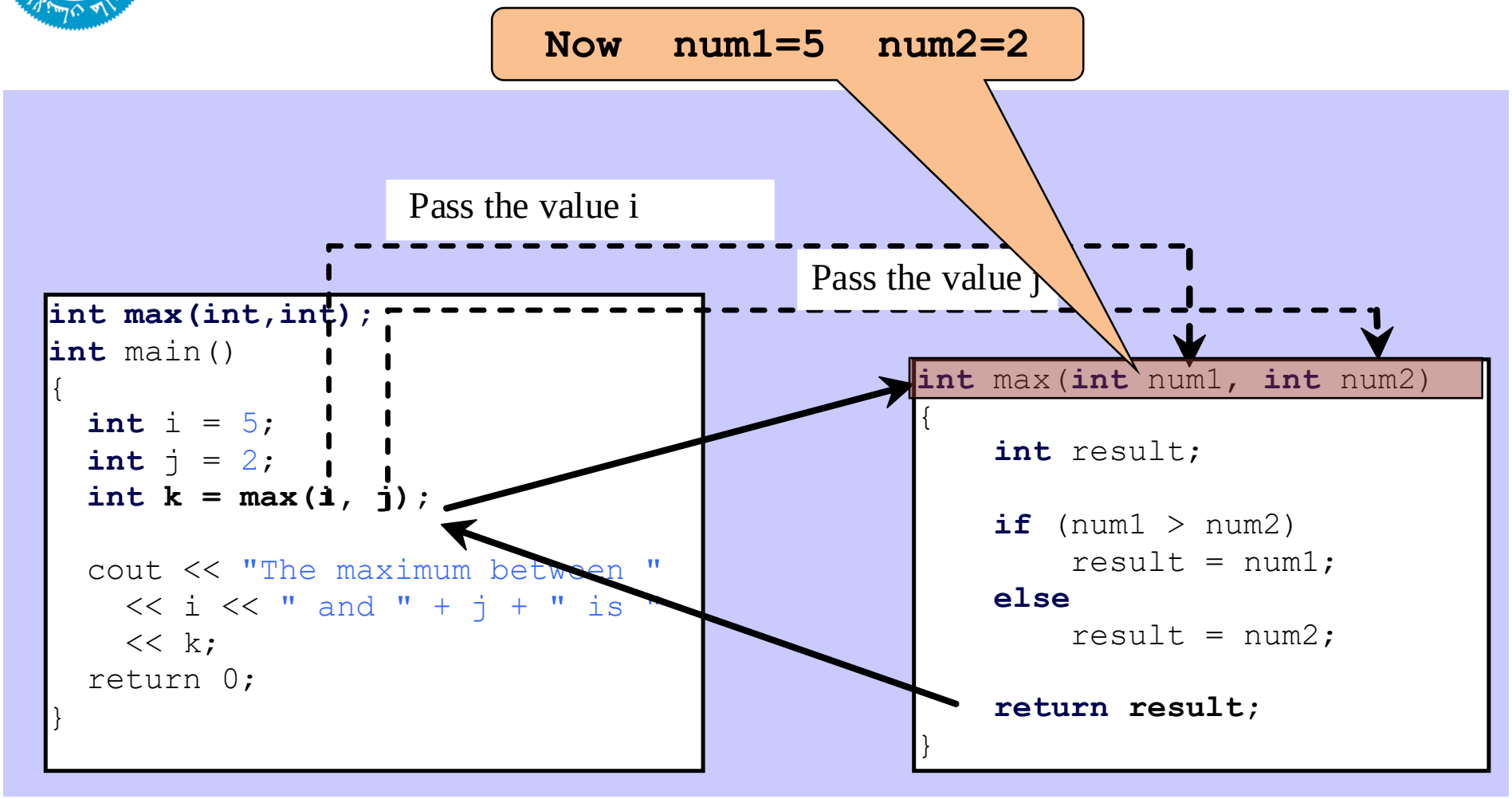
Calling Functions



Calling Functions

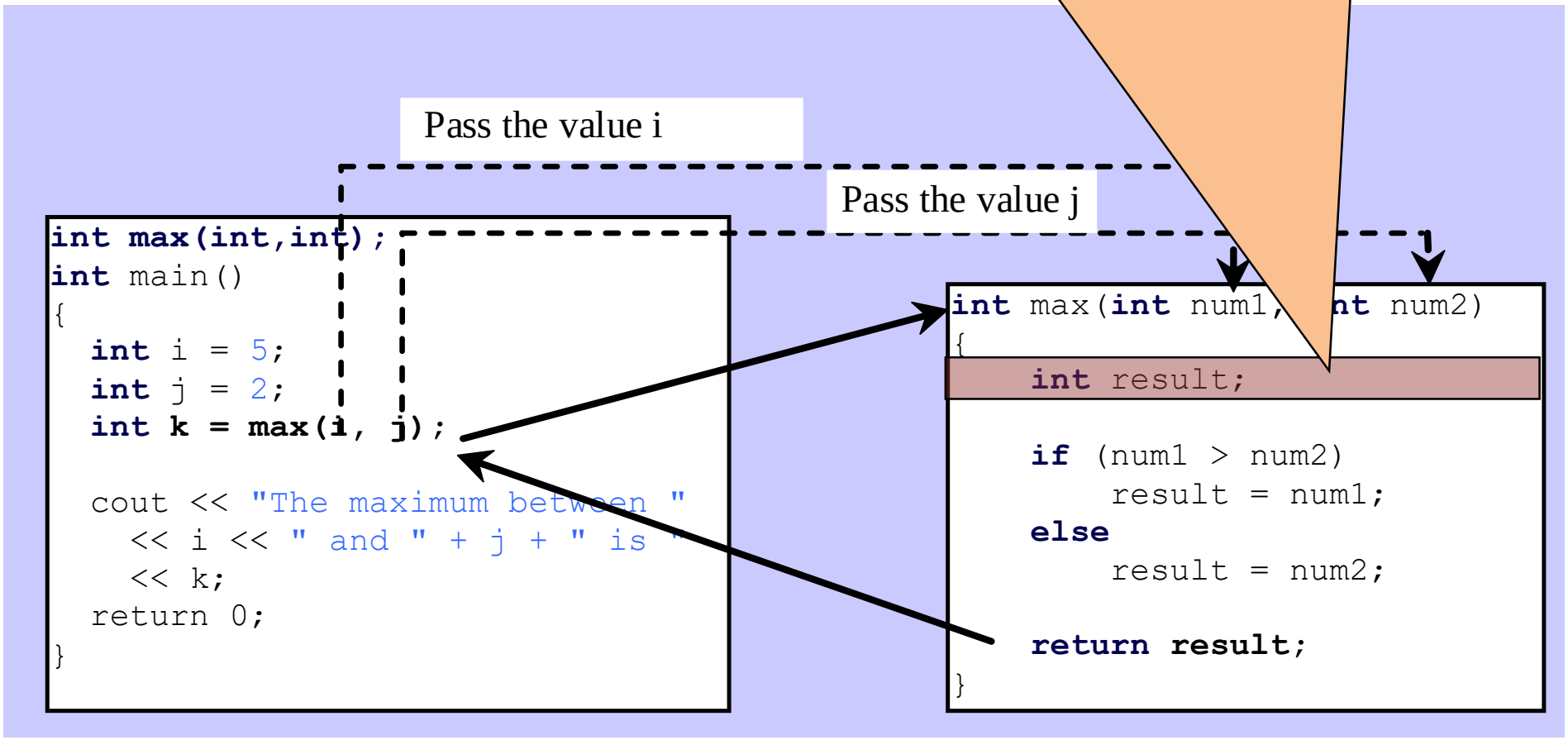


Calling Functions

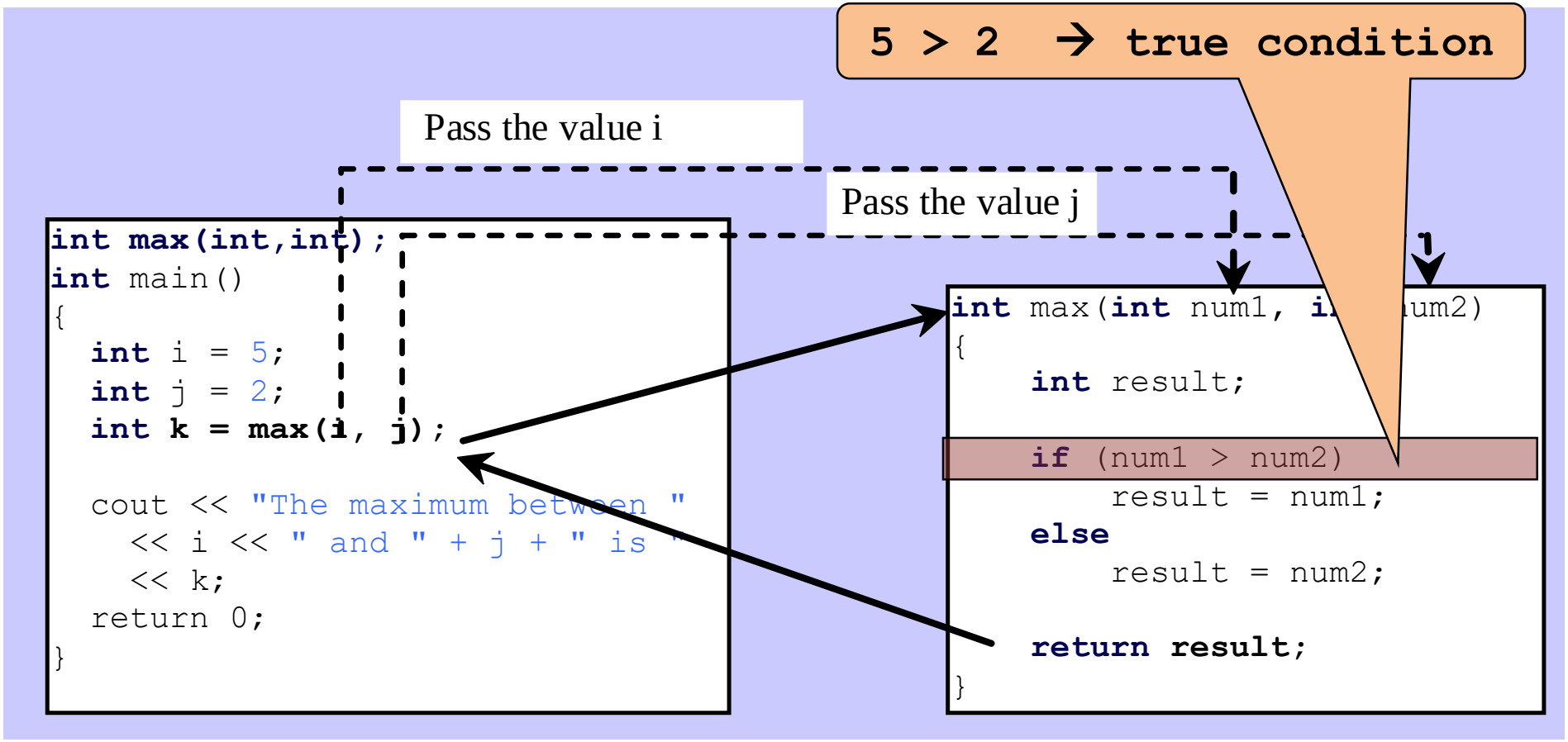


Calling Functions

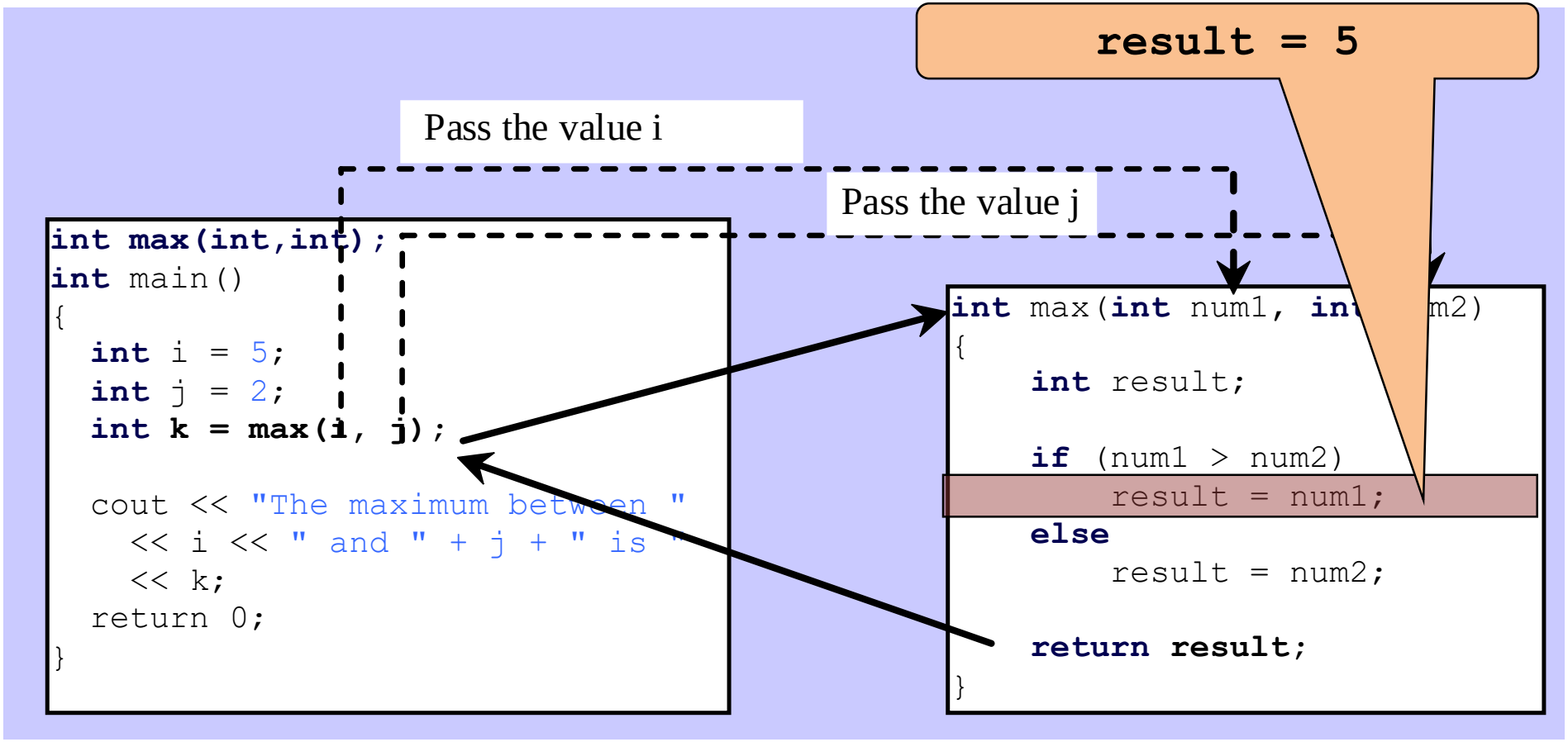
A new variable **result** will be created (local to the max function)



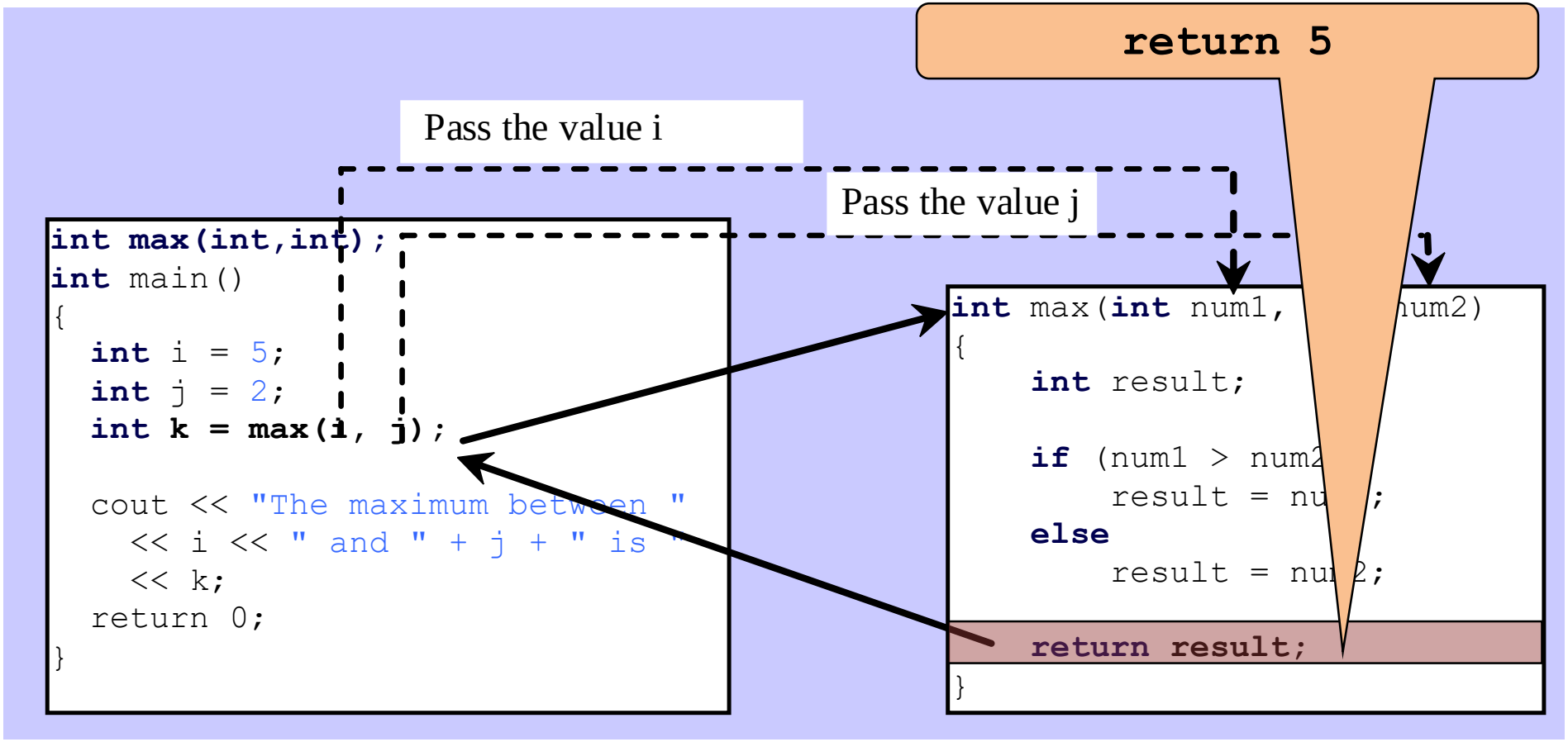
Calling Functions



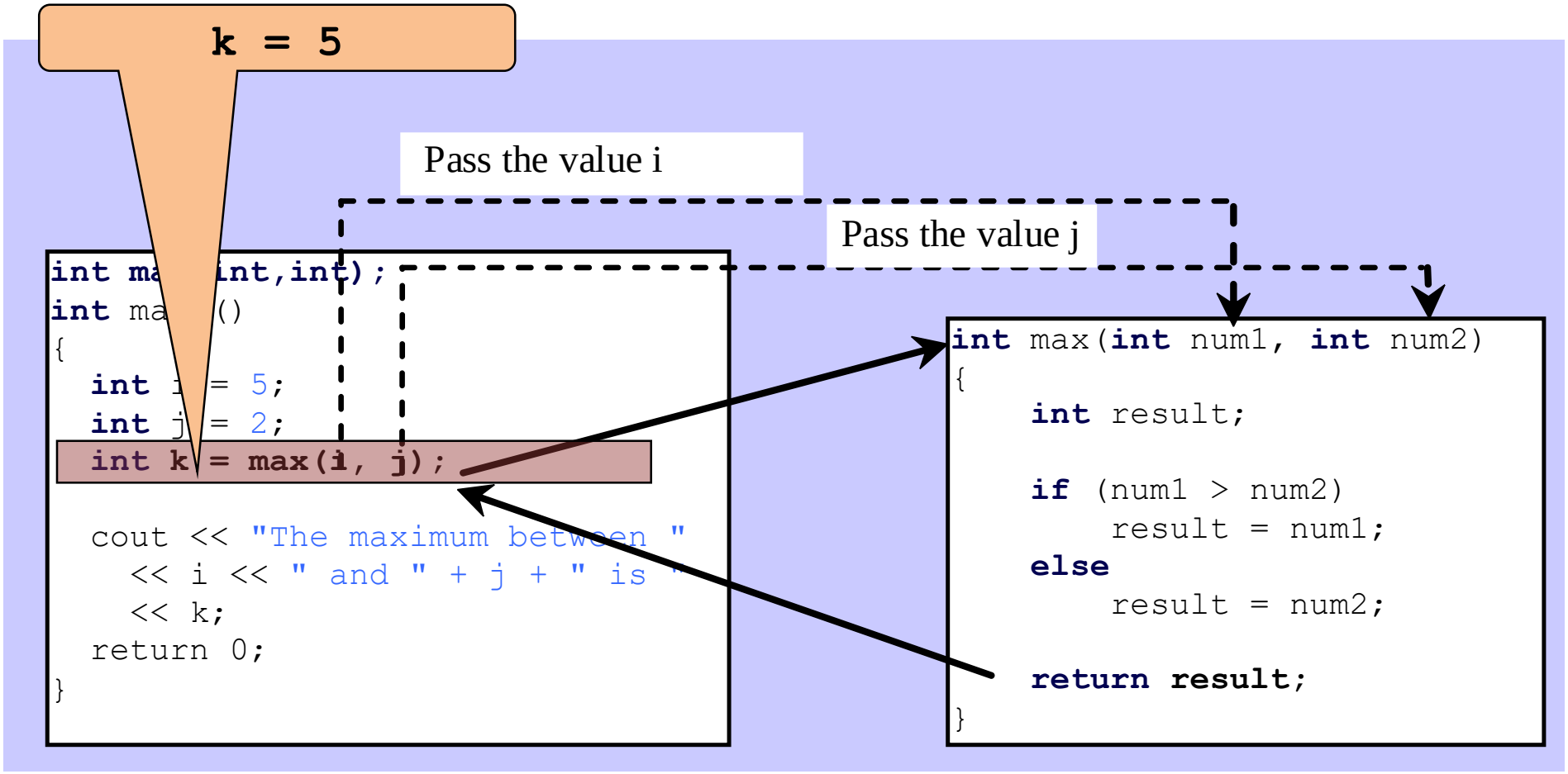
Calling Functions



Calling Functions



Calling Functions



Calling Functions

Display "The maximum between 5 and 2 is 5"

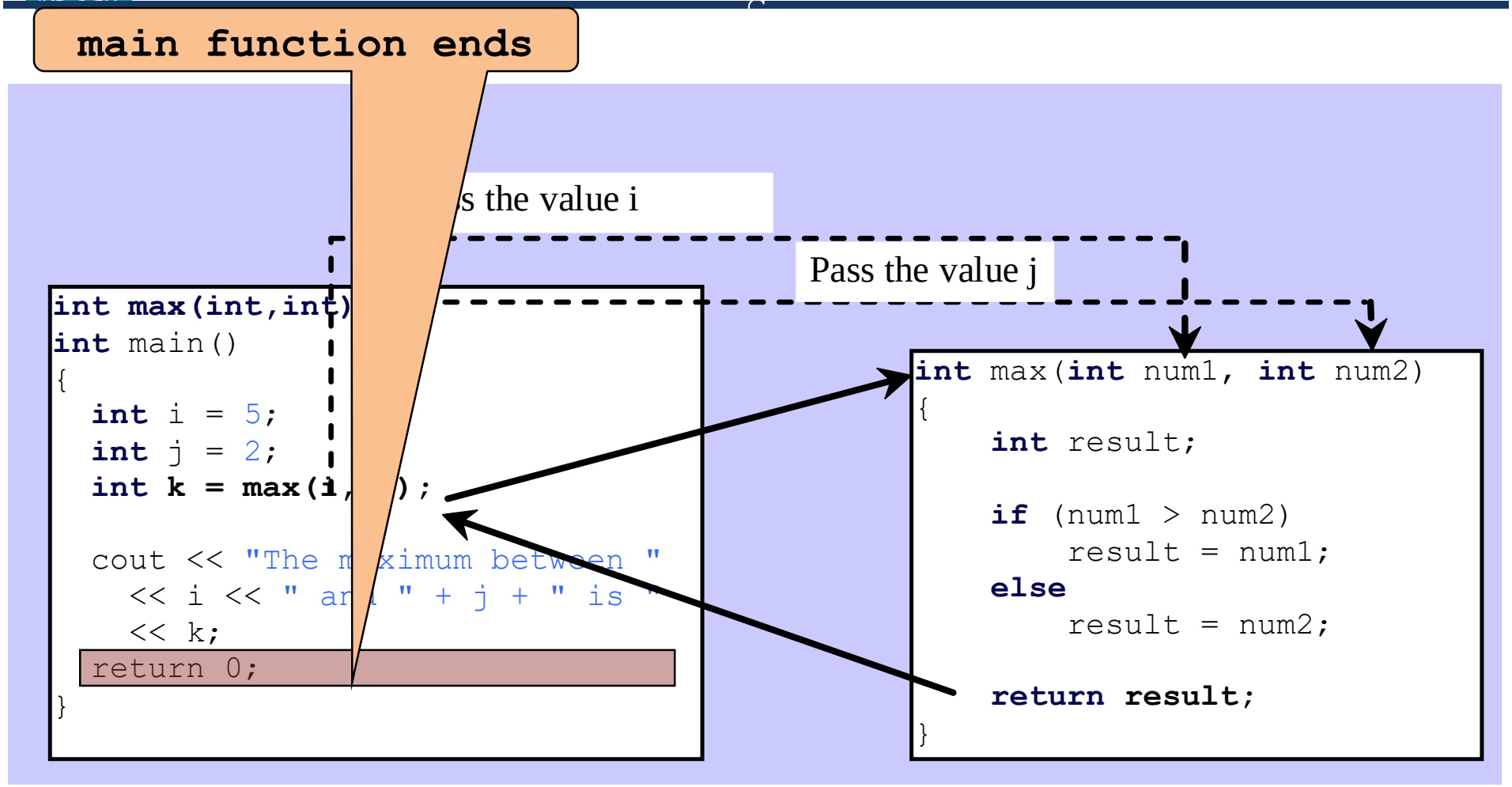
Pass the value i

Pass the value j

```
int max
int mai
{
  int i
  int j    2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int max(int num1, int num2)
{
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

Calling Functions



Function Overloading

- **Function overloading**
  - **Functions** with **same name** and **different parameters**
  - Should **perform similar tasks**:
    - i.e., function to square **int** and function to square **float** values

```
int square(int x)
{
    return (x * x);
}
```

```
float square(float x)
{
    return (x * x);
}
```

Function Overloading

- At call-time C++ complier selects the proper function by examining the number, type and order of the parameters

Function Overloading

```
void print(int  i)
{  cout << " Here is int " << i << endl;  }

void print(double f)
{ cout << " Here is float " << f << endl; }

void print(char c)
{ cout << " Here is char" << c << endl; }

int main()
{  print(10);   print(10.10);  print('Y'); }
```

Default Function Arguments

- A value auto assigned by compiler (if not provided by user)
- After default argument, all remaining function arguments must be default arguments
- Example….

Default Function Arguments - Example

```
int sum(int x, int y, int w=1, int z=2) {
      return (x+y+w+z);
}

int main() {
      cout<<sum(2,3);       //sum will be: 8
      cout<<sum(2,3,4);     //sum will be: 11
       cout<<sum(2,3,4,5); //sum will be: 14

      return 0;
}
```

Scope of a Variable

- The **scope of a variable**: the **part of the program** in which the **variable can be accessed**

- *Note: A variable cannot be used before it is defined*

- Example:…

Scope of a Variable

- Different levels of scope:
  1. Function scope
  2. block scope
  3. File scope
  4. Class scope

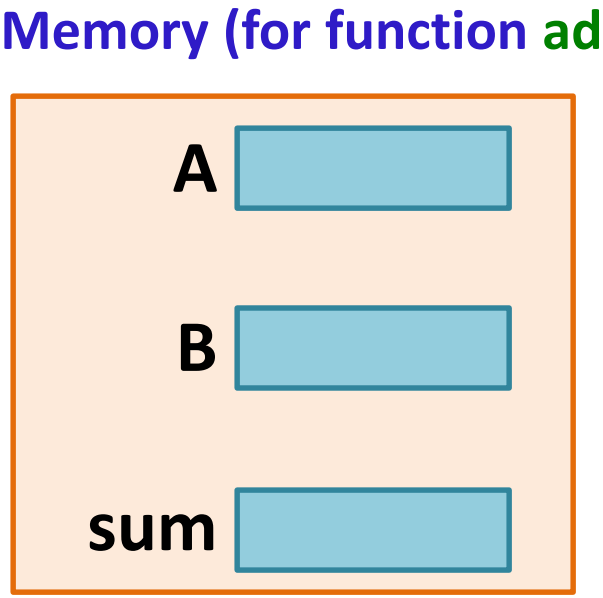
Local variables: 1, 2

Global variables: 3, 4

Scope of a Variable - Example

- **Formal parameters** and **variables** declared within **a function body** are **local** to that **function**:
  - *Cannot be accessed outside of that function*

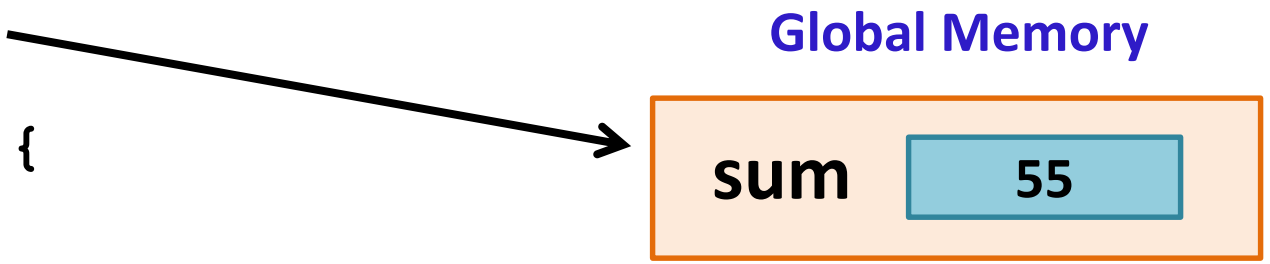
```
int add(int A, int B){
    int sum = a+b;
    return sum;
}
```



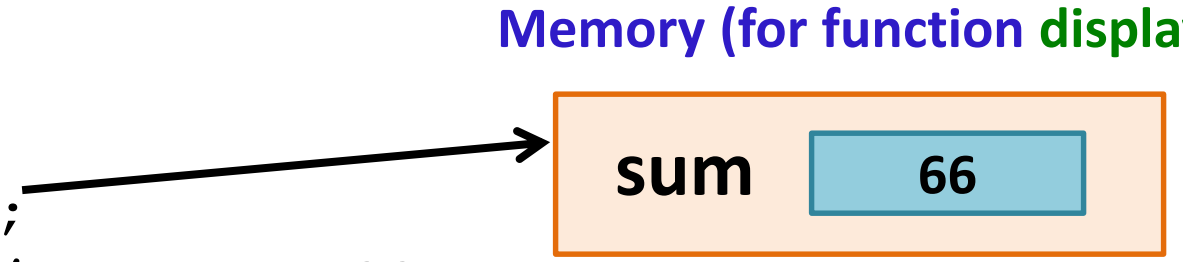
Scope of a Variable - Example

- **Global variables** with **same name**:

```
int sum=55;

Iint main() {

     …

}
```



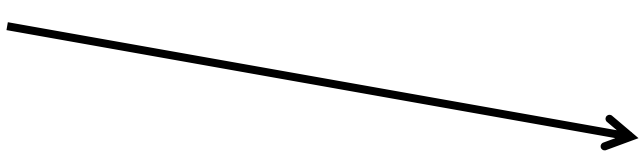

```
void display()

{
     int sum = 66;
     cout<<sum; // Display 66

}
```



Scope of a Variable - Example

- ***Global variables*** with **same name**:

```
int sum=55;

void main()

{

      …

}

void display()

{

      int sum = 66;

      cout<<::sum; // Display 55

}
```



Visibility of a Variable

A variable is ***visible*** within its scope, and ***invisible*** or ***hidden*** outside it.

Lifetime of a Variable

- The ***lifetime*** of a variable is the **interval of time in which storage is bound** to the variable.

- The **action that acquires storage** for a variable is called ***allocation***.

Lifetime of Variables

- **Local Variables** (**function** and **block** scope) have **lifetime** of the **function** or **block**

- **Global variable** (having **file level scope**) has **lifetime** until the **end of program**

- **Examples…**

Static Variables

Scope

- **Different levels of scope:**
  1. Function scope
  2. block scope

  } **Local variables**

  3. File scope

  } **Global variables**

  4. Class scope

Lifetime of Variables

- **Local Variables** (**function** and **block** scope) have **lifetime** of the **function** or **block**

- **Global variable** (having **file level scope**) has **lifetime** until the **end of program**

- **Examples…**

Static Variables

Static Variables:

- Is **created** at the **start of program** execution
- A **static variable** has **scope** of **local variable**
- But has **lifetime** of **global variables**
  - Therefore, **static variables retain** their **contents** or **values** (***until the program ends***)
- **If not initialized**, it is **assigned value 0** (*automatically by the compiler*)

Static Variables - Example

- In the following example, the **static variable *sum*** is **initialized to 1**

```
static int sum = 1;
```

- **Initialization** takes place **only once**.
- If declaration **in** a **user-defined function**:
  - **First time** the function is called, the **variable sum** is **initialized to 1**.
  - **Next time** the **function** (containing the above declaration) is **executed**, but sum is **not reset to 1**.

Home Exercise (Using Static Variable)

- Write a function that, when you call it, displays a message telling how many times it has been called: “I have been called 3 times”, for instance. Write a main() program that ask the user to call the function, if the user presses ‘y’ the function is called otherwise if ‘n’ is pressed the program terminates.

NOTE: Do not use any global variable or pass any value in the function