Scope and Parameter Passing

Lecture 17

Sections 6.5, 6.10, 6.13

Robb T. Koether

Hampden-Sydney College

Fri, Oct 4, 2019

Outline

The Scope of an Object

- The scope of an object is the part of the program over which that object is recognized.
- The scope of an object may be
  - Statement scope – A single statement.
  - Block scope – A block within a function.
  - Function scope – A function.
  - Class scope – A class (to be discussed soon).
  - Global scope – The entire program.
- An object cannot be accessed outside of its scope.

The Scope of an Object

- When a variable passes out of scope, it ceases to exist.
- The memory that was allocated to it is deallocated.
- A variable whose scope is the entire program is called global.
- All other variables are local (to the statement, block, function, etc.).

Outline

Statement Scope

- A volatile, or unnamed, object is an object that occurs as the value of an expression or a subexpression.
- For example, `a + b` is a volatile object in the statement

```
c = a + b;
```

- The scope of a volatile object is the statement in which it occurs.
- When the statement is completely evaluated, the volatile object passes out of scope.

Statement Scope

Statement Scope

```
int a = 2;
int b = 3;
int c = a + b;  // a + b has stmt scope
```

- Variables `a`, `b`, and `c` are named objects and are stored in memory.
- Expression `a + b` (with value `5`) is unnamed and has only statement scope.

Outline

Block Scope

- If an object is declared within a block `{ }` within a function, its scope is limited to that block.
- The scope extends from the point of declaration to the end of the block in which the variable is declared.
- When execution leaves the block, the object passes out of scope.

Block Scope

Block Scope

```
if (a > b)
{
    int max = a;     // Block scope
}
else
{
    int max = b;     // Block scope
}
cout << "The max is " << max << endl;
```

- This will not work. Why?

Block Scope

Block Scope

```
if (a > b)
    int max = a;
else
    int max = b;
cout << "The max is " << max << endl;
```

- This will not work either. Why?

Block Scope

Block Scope

```
int max;          // Function scope
if (a > b)
    max = a;
else
    max = b;
cout << "The max is " << max << endl;
```

- This will work. Why?

Outline

Function Scope

- Variables declared within a function are recognized only within that function.
- The scope extends from the point of declaration to the end of the function.
- When execution returns from the function, the object passes out of scope.

Function Scope

Function Scope

```
int max(int a, int b)
{
    int larger;        // Function scope
    if (a > b)
        larger = a;
    else
        larger = b;
    return larger;
}
```

- The variable `larger` is not accessible from anywhere outside of this function.

Function Scope

- Variables declared within a function are recognized only within that function.
- The scope extends from the point of declaration to the end of the function.
- When execution returns from the function, the object passes out of scope.

Function Scope

Function Scope

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

- This does the same thing, but is more concise.

Function Scope

Function Scope

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

- This also does the same thing, but maybe is too concise.
- It's kinda hard to read.

Outline

Global Scope

- A variable is global if it is declared outside of any function.
- Global variables are recognized in all functions.
- Good programming practice strongly discourages the use of global variables.

Examples of Scope

- Example
  - `ScopeExample.cpp`

Outline

Methods of Passing Parameters

- Parameters may be passed by one of two methods.
  - By value.
  - By reference.

Outline

Value Parameters

- When an object is passed by value, the value of the actual parameter is *copied* to the formal parameter.
- The function uses the local copy of the parameter.
- The copy constructor is used to make the copy (to be discussed later).
- The actual parameter is left unchanged, no matter what is done to the local copy.

Value Parameters

```
int main()
{
        :
    f(a, b)
        :
}
```

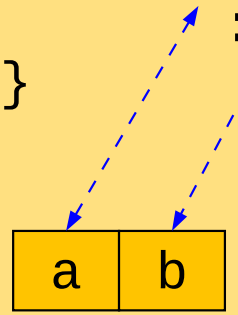
a	b

```
int f(int c, int d)
{
        :
    sum = c + d;
        :
}
```

`a` and `b` are local to `main()`

Value Parameters

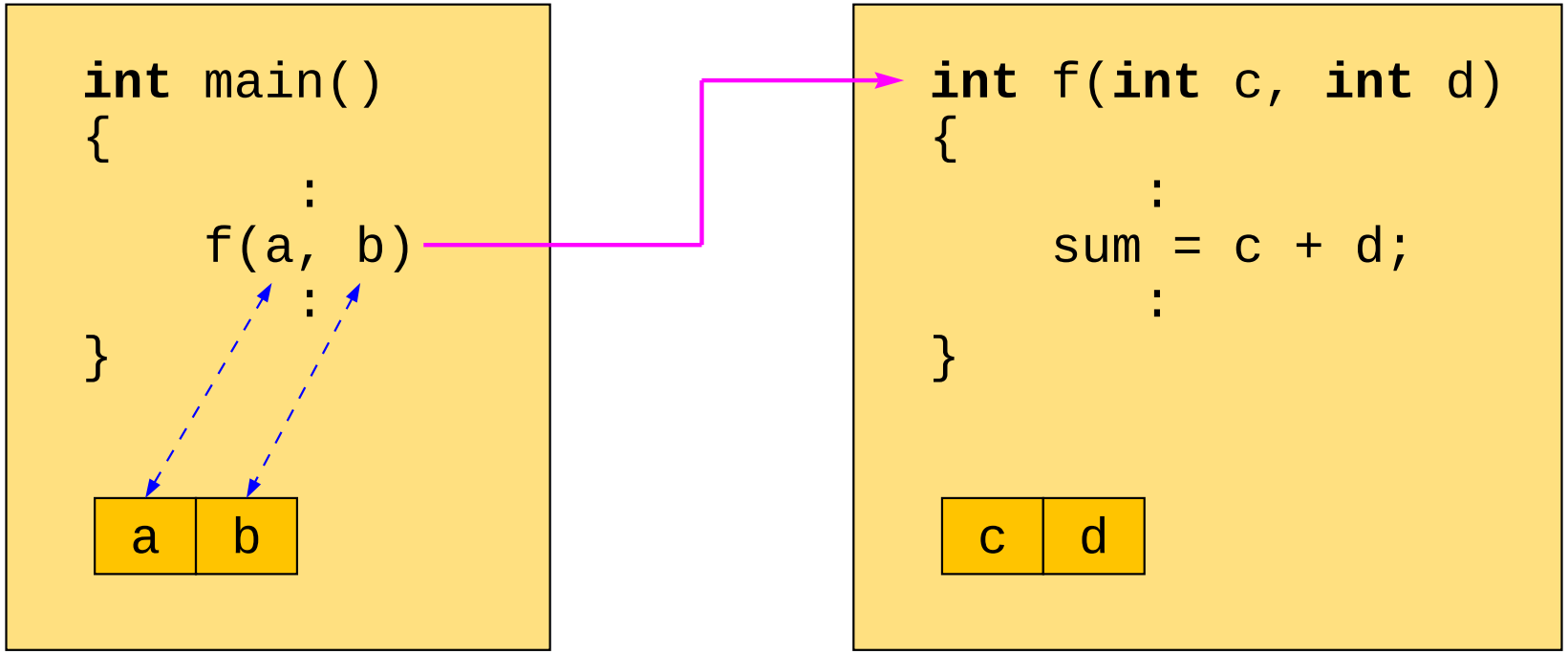
```
int main()
{
        :
    f(a, b)
        :
}
```




```
int f(int c, int d)
{
        :
    sum = c + d;
        :
}
```

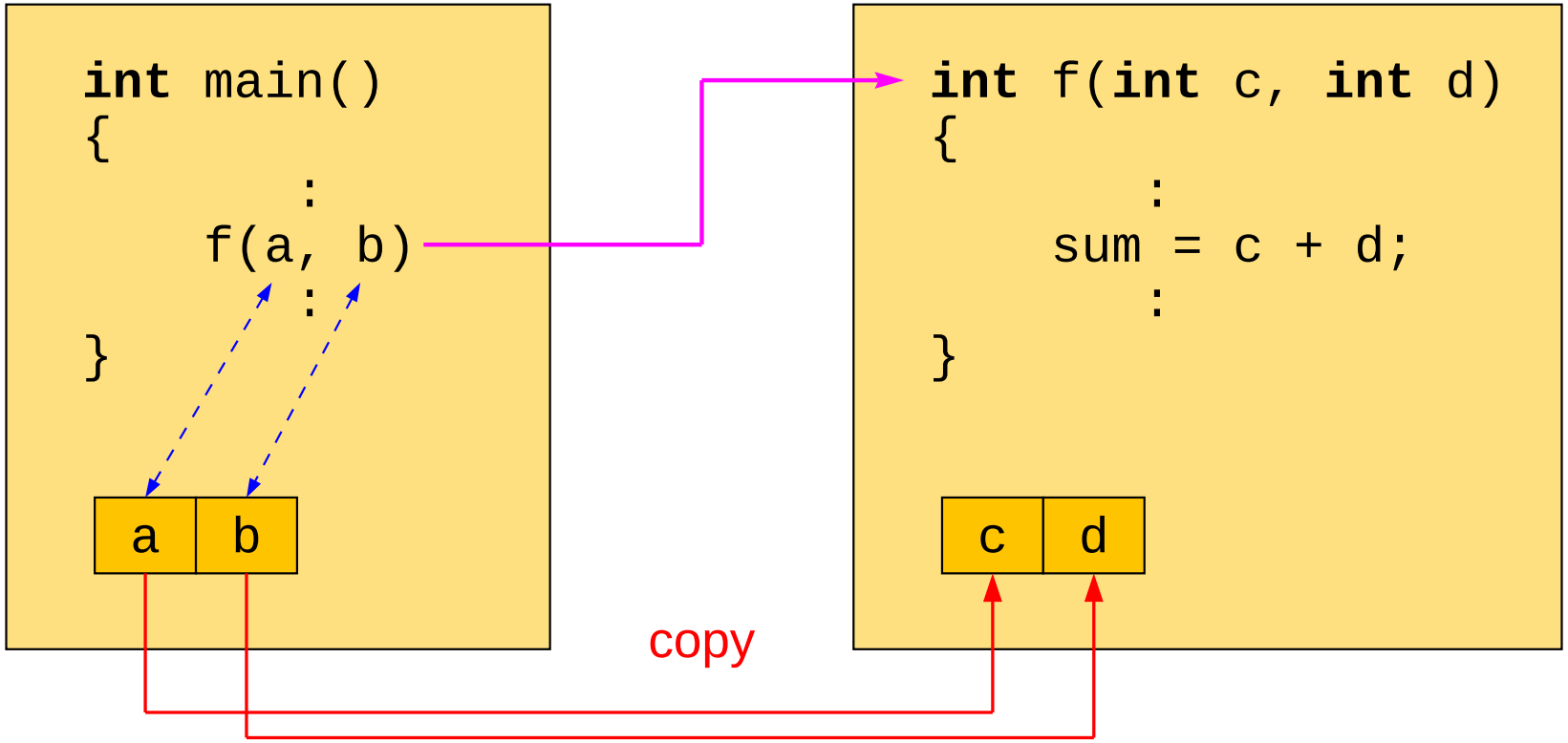
Their values are stored locally

Value Parameters



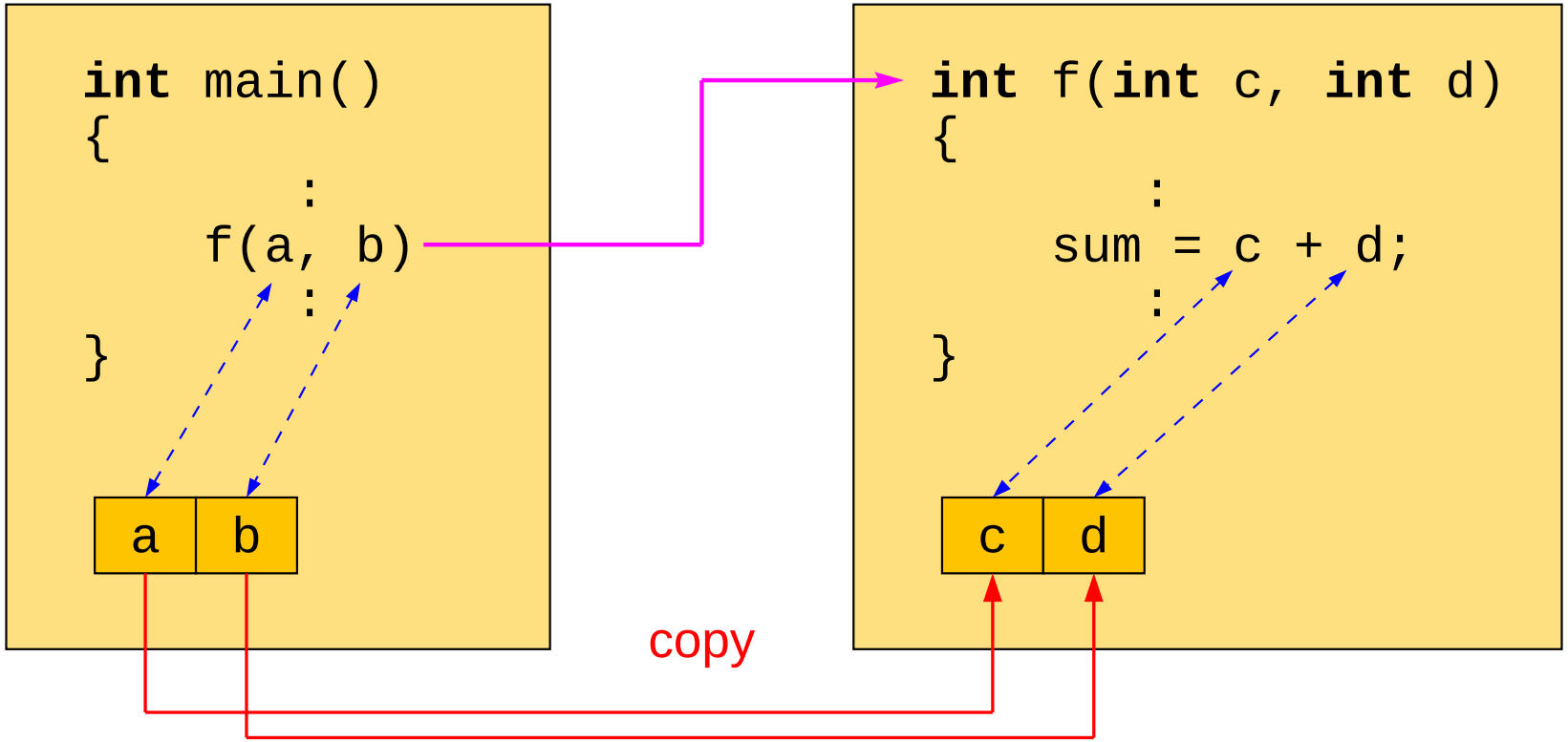
Memory is allocated in `f()` for the copies

Value Parameters



Copies of `a` and `b` are made in `f()`

Value Parameters



`c` and `d` are local to `f()`

Outline

Reference Parameters

- When an object is passed by reference, the *address* of the actual parameter is passed to the function.
- The formal parameter becomes a reference to the actual parameter, i.e., the actual parameter and the formal parameter are the same object.
- If the formal parameter is changed, so is the actual parameter.
- An ampersand `&` is used in the formal parameter list to indicate a reference parameter.

Reference Parameters

```
int main()
{
        :
    f(a, b)
        :
}
```

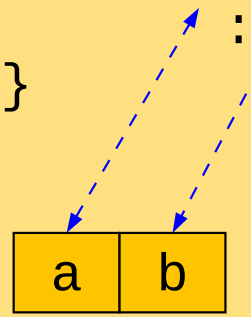
a	b

```
int f(int& c, int& d)
{
        :
    sum = c + d;
        :
}
```

`a` and `b` are local to `main()`

Reference Parameters

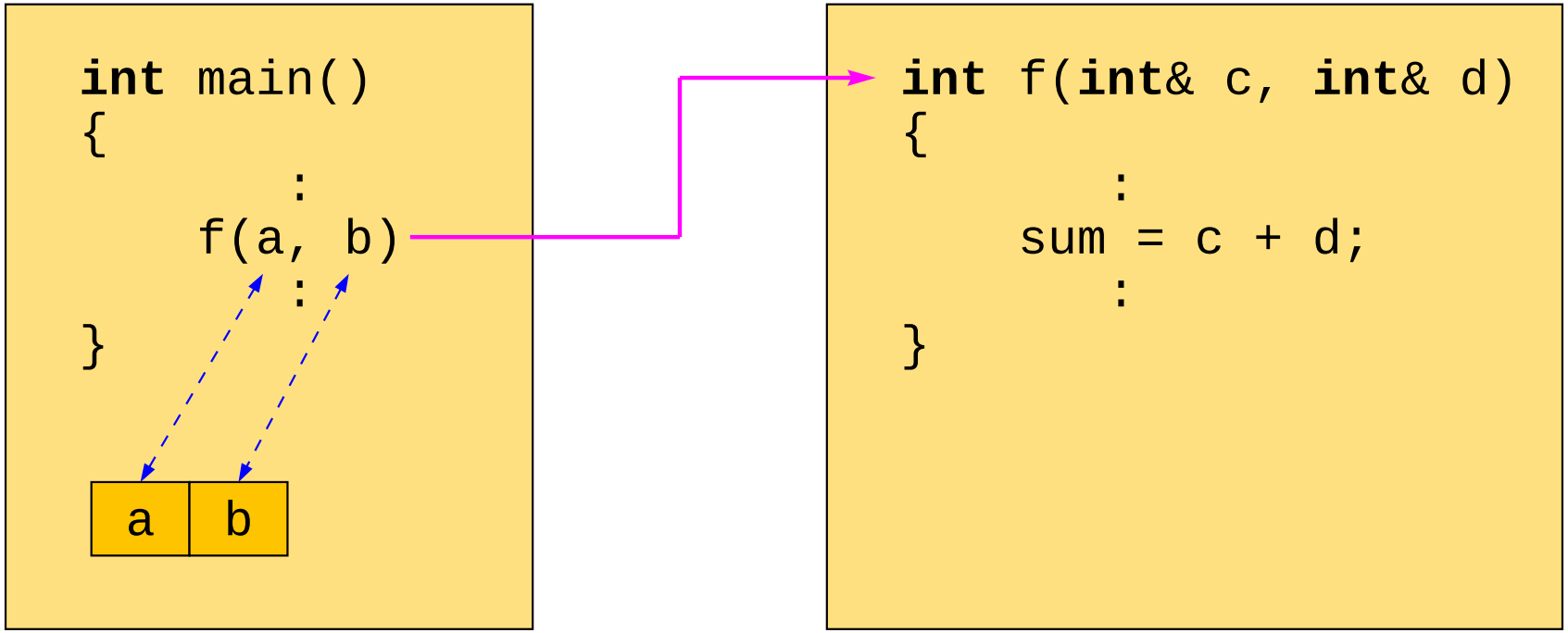
```
int main()
{
        :
    f(a, b)
        :
}
```




```
int f(int& c, int& d)
{
        :
    sum = c + d;
        :
}
```

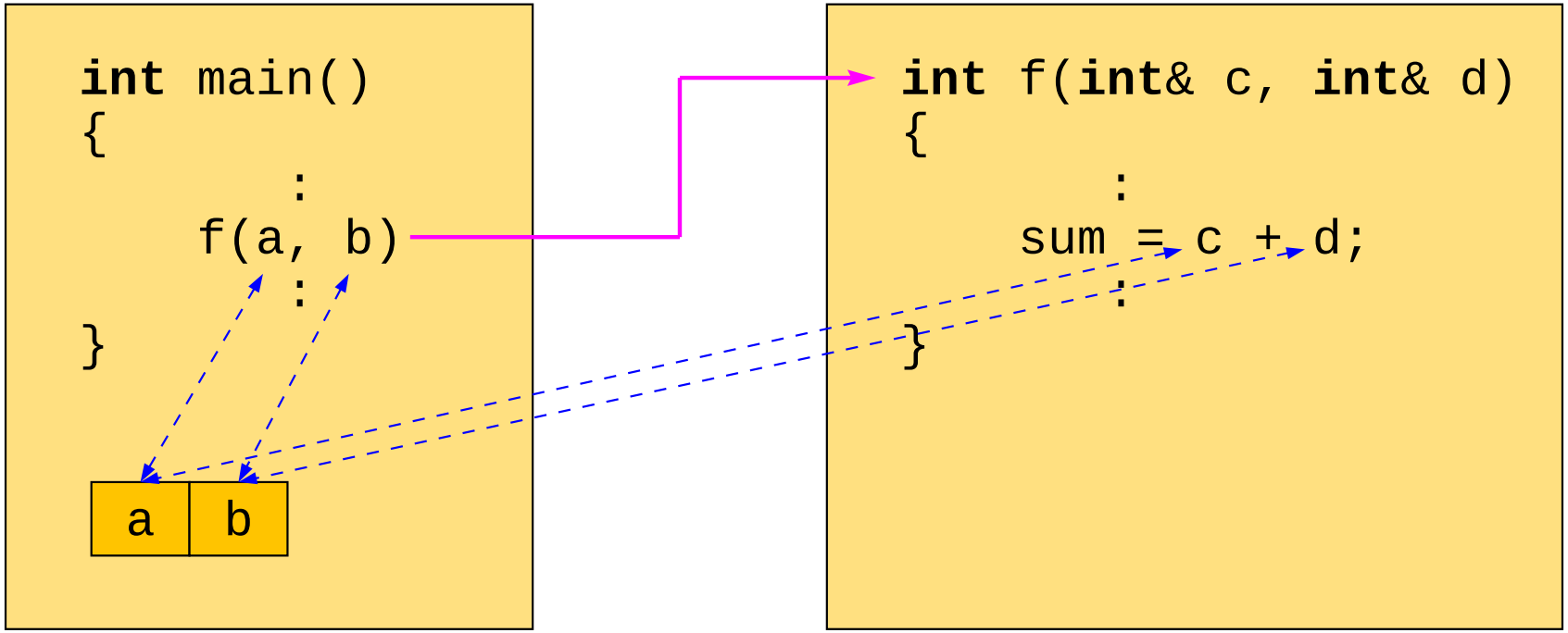
Their values are stored locally

Reference Parameters



No memory is allocated in `f()`

Reference Parameters



`c` and `d` refer to `a` and `b` in `main()`

Outline

Examples of Parameter Passing

- Examples
  - `GCDExample.cpp`
  - `GetWordFunc.cpp`

Outline

Assignment

Assignment

- Read Sections 6.5, 6.10, 6.13.