# Functions and Scopes

Sai-Keung Wong

National Yang Ming Chiao Tung University

Hsinchu, Taiwan

# Intended Learning Outcomes

- Describe functions and scopes
- Describe the process of function call
- Distinguish between formal parameters and actual parameters
- Distinguish between pass-by-value and pass-by-reference
- List the benefits of function abstraction

# Content

- Function declaration
- Formal parameters
- Actual parameters (or arguments)
- Function invocation
- Pass-by-value
- Pass-by-reference
- Stack content when a function is invoked (Call stack)
- Overloading functions
- Ambiguous invocation
- Function prototypes

- Scopes
- Static local variables
- Global variables
- Unary scope resolution ::
- Constant reference parameters
- System design

  stepwise refinement

# Function examples

Performing a single task

# Example

For each question, implement one function.

1. Input two numbers x and y

2. Generate randomly the two numbers

# Example

For each question, implement one function.

1. Input two numbers x and y

2. Generate randomly the two numbers

```
//input two numbers
void f( int &x, int &y ) {    // pass by reference
            cin >>  x >> y;
}
//generate two numbers randomly
void g( int &x, int &y ) {    // pass by reference
            x = rand( );
            y = rand( );
}
```

# Example

For each question, implement one function.

1. Input two numbers x and y
2. Generate randomly the two numbers

```
//input two numbers
void f( int &x, int &y ) {    // pass by reference
            cin >>  x >> y;
}
//generate two numbers randomly
void g( int &x, int &y ) {    // pass by reference
            x = rand( );
            y = rand( );
}
```

```
void test( ) {
   int a, b;
   f( a, b );
   g( b, a );
}
```

# Example

For each question, implement one function.

1. Input two numbers x and y
2. Generate randomly the two numbers

How do we know that each function is correct?

```
//input two numbers
void f( int &x, int &y ) {    // pass by reference
            cin >>  x >> y;
}
//generate two numbers randomly
void g( int &x, int &y ) {    // pass by reference
            x = rand( );
            y = rand( );
}
```

```
void test( ) {
   int a, b;
   f( a, b );
   g( b, a );
}
```

# Example

For each question, implement one function.

1. Input two numbers x and y
2. Generate randomly the two numbers

How do we know that each function is correct?

```
void test( ) {
   int a, b;
   f( a, b );
   cout << “function f” << endl;
   cout << “a:” << a << “\t” << “b:” << b << endl;
   g( b, a );
   cout << “function g” << endl;
   cout << “a:” << a << “\t” << “b:” << b << endl;
}
```

```
//input two numbers
void f( int &x, int &y ) {    // pass by reference
            cin >>  x >> y;
}
//generate two numbers randomly
void g( int &x, int &y ) {    // pass by reference
            x = rand( );
            y = rand( );
}
```

```
void test( ) {
   int a, b;
   f( a, b );
   g( b, a );
}
```

# Example

For each question, implement one function.

1. Input two numbers x and y

2. Generate randomly the two numbers

Show the result.

```
void test( ) {
   int a, b;
   f( a, b );
   cout << “function f” << endl;
   cout << “a:” << a << “\t” << “b:” << b << endl;
   g( b, a );
   cout << “function g” << endl;
   cout << “a:” << a << “\t” << “b:” << b << endl;
}
```

```
//input two numbers
void f( int &x, int &y ) {     // pass by reference
            cin >>  x >> y;
}
//generate two numbers randomly
void g( int &x, int &y ) {     // pass by reference
            x = rand( );
            y = rand( );
}
```

```
void test( ) {
   int a, b;
   f( a, b );
   g( b, a );
}
```

Do we need to set the seed?

# Example

For each question, implement one function.

1. Show the values of x and y
2. Show their sum
3. Swap the two variables

```
void test( ) {
  int a, b;
  a = 4; b = 7;
  showValues( a, b );
  showSum( a, b );
  swap( a, b );
}
```

```
void showValues( int x, int y ) {     // pass by value
            cout << x << “\t” <<y << endl;
}

void showSum( int x, int y ) {        // pass by value
            cout << x + y << endl;
}

void swap( int &x, int &y) {          // pass by reference
            int tmp;
            tmp = x; x = y; y = tmp;
}
```

# Example

For each question, implement one function.

1. Show the values of x and y
2. Show their sum
3. Swap the two variables

```
void test( ) {
   int a, b;
   a = 4; b = 7;
   showValues( a, b );
   showSum( a, b );
   swap( a, b );
}
```

```
void showValues( int x, int y ) {     // pass by value
             cout << x << “\t” <<y << endl;
}

void showSum( int x, int y ) {        // pass by value
             cout << x + y << endl;
}

void swap( int &x, int &y) {          // pass by reference
             int tmp;
             tmp = x; x = y; y = tmp;
}
```

How do we know that each function is correct?

**Show the result.**

# Function definition

breakdown

# Function definition

A function is a A1 of statements that are A2 together to perform an A3

```
int min( int num1, int num2 )
{
    int min_value;
    min_value = num1;
    if ( num2 < num1 )
        { min_value = num2; }
    return min_value;
}
```

# Function definition

A function is a collection of statements that are grouped together to perform an operation.

```
int min( int num1, int num2 )
{
    int min_value;
    min_value = num1;
    if ( num2 < num1 )
        { min_value = num2; }
    return min_value;
}
```

# Function definition

A function is a collection of statements that are grouped together to perform an operation.

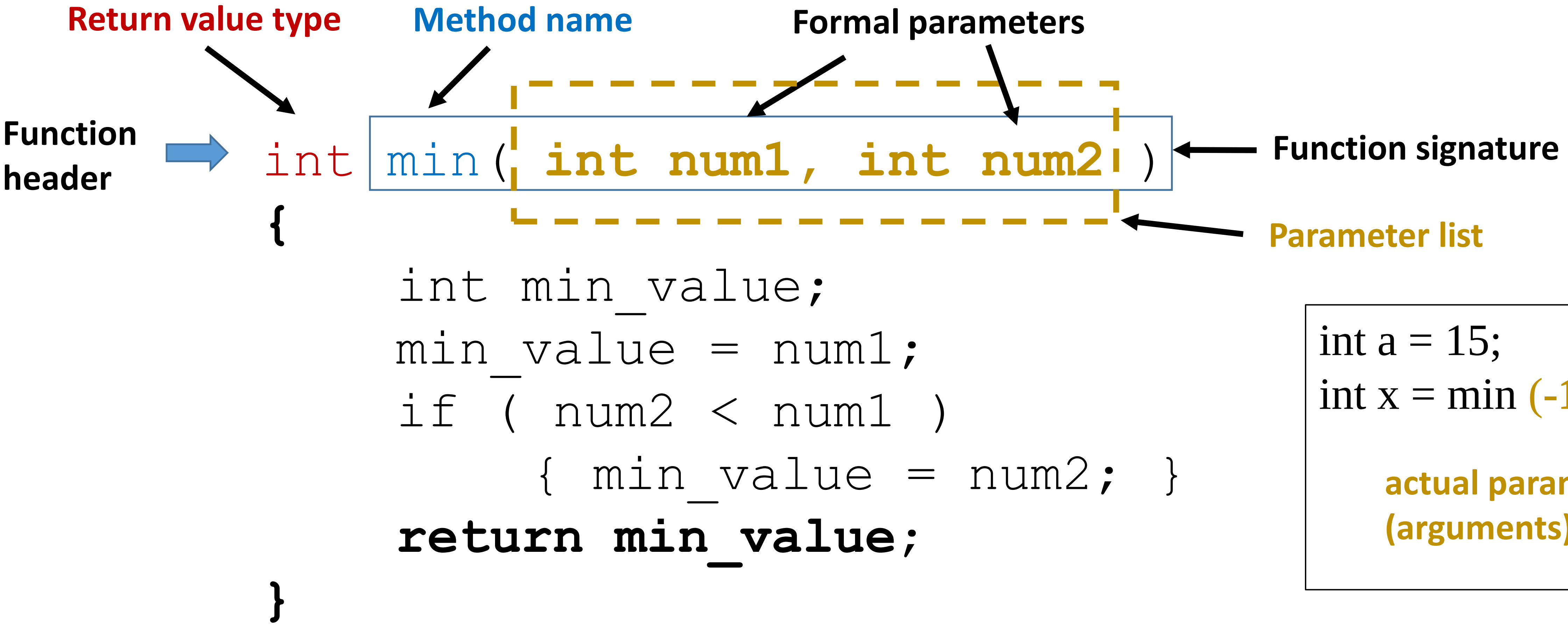
```
int min( int num1, int num2 )
{
        int min_value;
        min_value = num1;
        if ( num2 < num1 )
                { min_value = num2; }
        return min_value;
}
```

```
int a = 15;
int x = min (-12, a);
```

**actual parameters (arguments)**

# Function definition

A function is a collection of statements that are grouped together to perform an operation.



Return value type

Method name

Formal parameters

Function header

Function signature

Parameter list

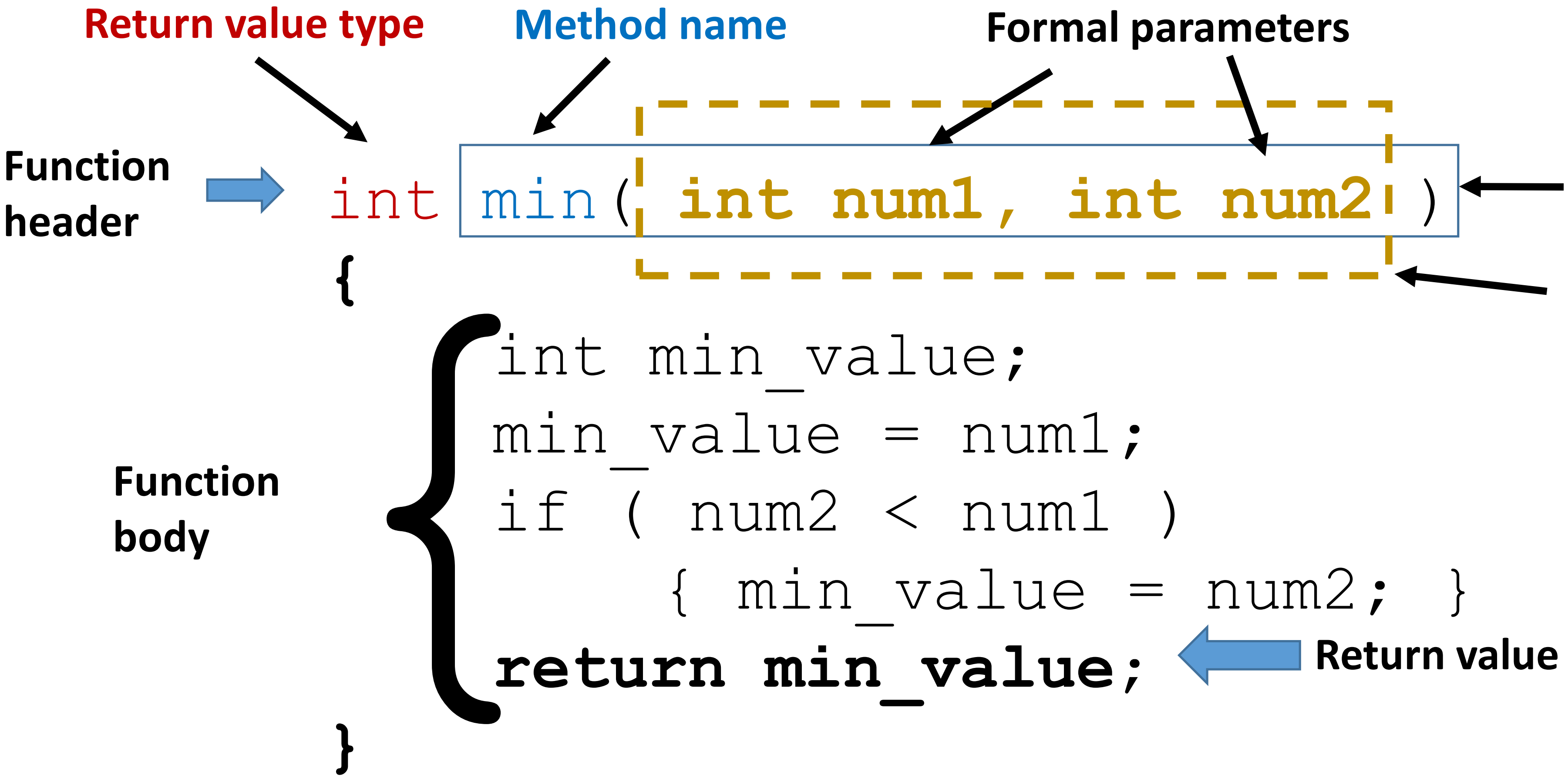
```
int min( int num1, int num2 )
{
      int min_value;
      min_value = num1;
      if ( num2 < num1 )
           { min_value = num2; }
      return min_value;
}
```

```
int a = 15;
int x = min (-12, a);
```

actual parameters (arguments)

# Function definition

A function is a collection of statements that are grouped together to perform an operation.



Return value type

Method name

Formal parameters

Function header

Function signature

Parameter list

Function body

Return value

```
int min( int num1, int num2 )
{
    int min_value;
    min_value = num1;
    if ( num2 < num1 )
         { min_value = num2; }
    return min_value;
}
```

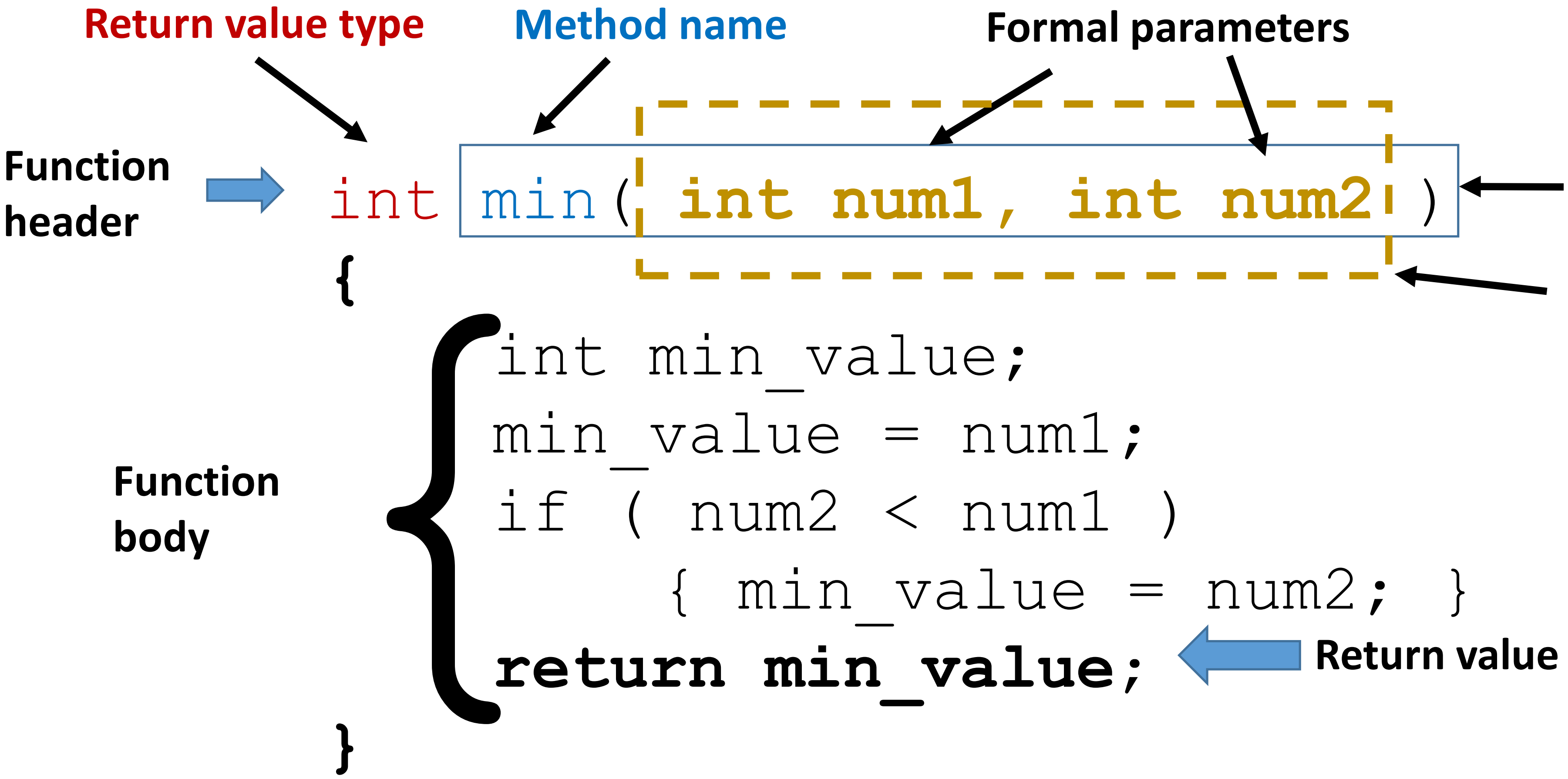
```
int a = 15;
int x = min (-12, a);
```

actual parameters (arguments)

**Function Body: the part enclosed by the outer most braces { and }.**

# Function definition

A function is a collection of statements that are grouped together to perform an operation.



Return value type

Method name

Formal parameters

Function header

Function signature

Parameter list

Function body

Return value

```
int min( int num1, int num2 )
{
    int min_value;
    min_value = num1;
    if ( num2 < num1 )
        { min_value = num2; }
    return min_value;
}
```

```
int a = 15;
int x = min (-12, a);
```

actual parameters (arguments)

**Function Body: the part enclosed by the outer most braces { and }.**

# Function definition

A function is a collection of statements that are grouped together to perform an operation.

```
int min( int num1, int num2 )
{
    int min_value;
    min_value = num1;
    if ( num2 < num1 )
         { min_value = num2; }
    return min_value;
}
```

- A function may return a A1
- The returnValueType is the A2 of the value the function returns.
- If the function does not return a value, the returnValueType is A3
- *void* is a A4

# Function definition

A function is a collection of statements that are grouped together to perform an operation.

```
int min( int num1, int num2 )
{
     int min_value;
     min_value = num1;
     if ( num2 < num1 )
          { min_value = num2; }
     return min_value;
}
```

- A function may return a value.
- The returnValueType is the data type of the value the function returns.
- If the function does not return a value, the returnValueType is *void*.
- *void* is a keyword.

A1

```
k( ) {
  cout << “nothing” << endl;
}
```

# Function definition

A function is a collection of statements that are grouped together to perform an operation.

```
int min( int num1, int num2 )
{
     int min_value;
     min_value = num1;
     if ( num2 < num1 )
          { min_value = num2; }
     return min_value;
}
```

- A function may return a value.
- The returnValueType is the data type of the value the function returns.
- If the function does not return a value, the returnValueType is *void*.
- *void* is a keyword.

```
void k( ) {
  cout << “nothing” << endl;
}
```

# Actual and formal parameters

- Actual parameters: appear in function calls
- Formal parameters: appear in function declarations

```
void k( int a, int &b ) {
            ……
}
void g( int n ) {
            k(10, n);
}
int main( int argc, char **argv ) {
            g( 5 );
            return 0;
}
```

# Actual and formal parameters

- Actual parameters: appear in function calls
- Formal parameters: appear in function declarations

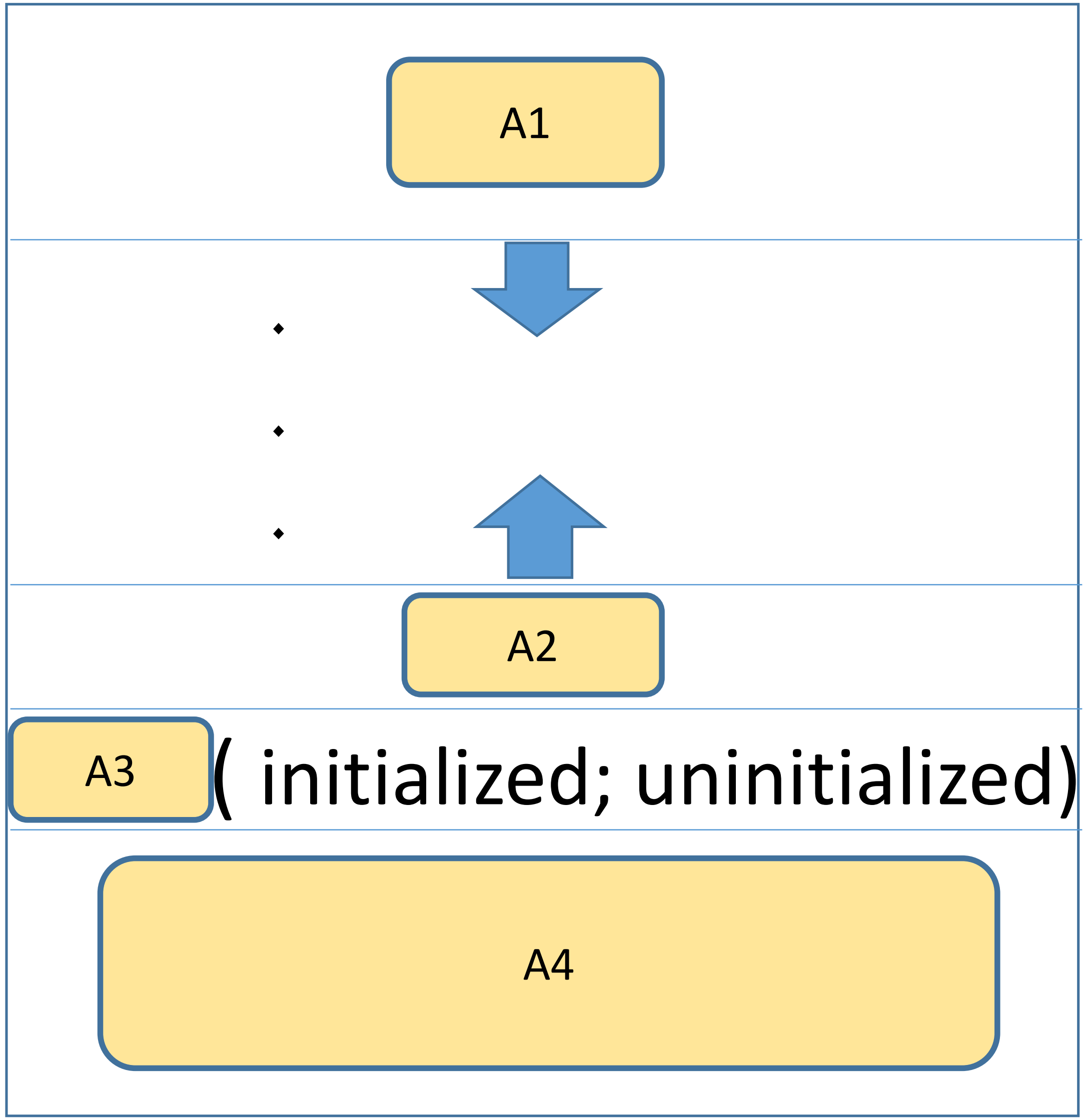
formal or actual?

```
void k( int a, int &b ) {               // a and b are A1 parameters
        ......
}
void g( int n ) {                       // n is a A2 parameter
        k(10, n);                       // 10 and n are A3 parameters
}
int main( int argc, char **argv ) {     // argc and argv are A4 parameters
        g( 5 );                         // 5 is an A5 parameter
        return 0;
}
```

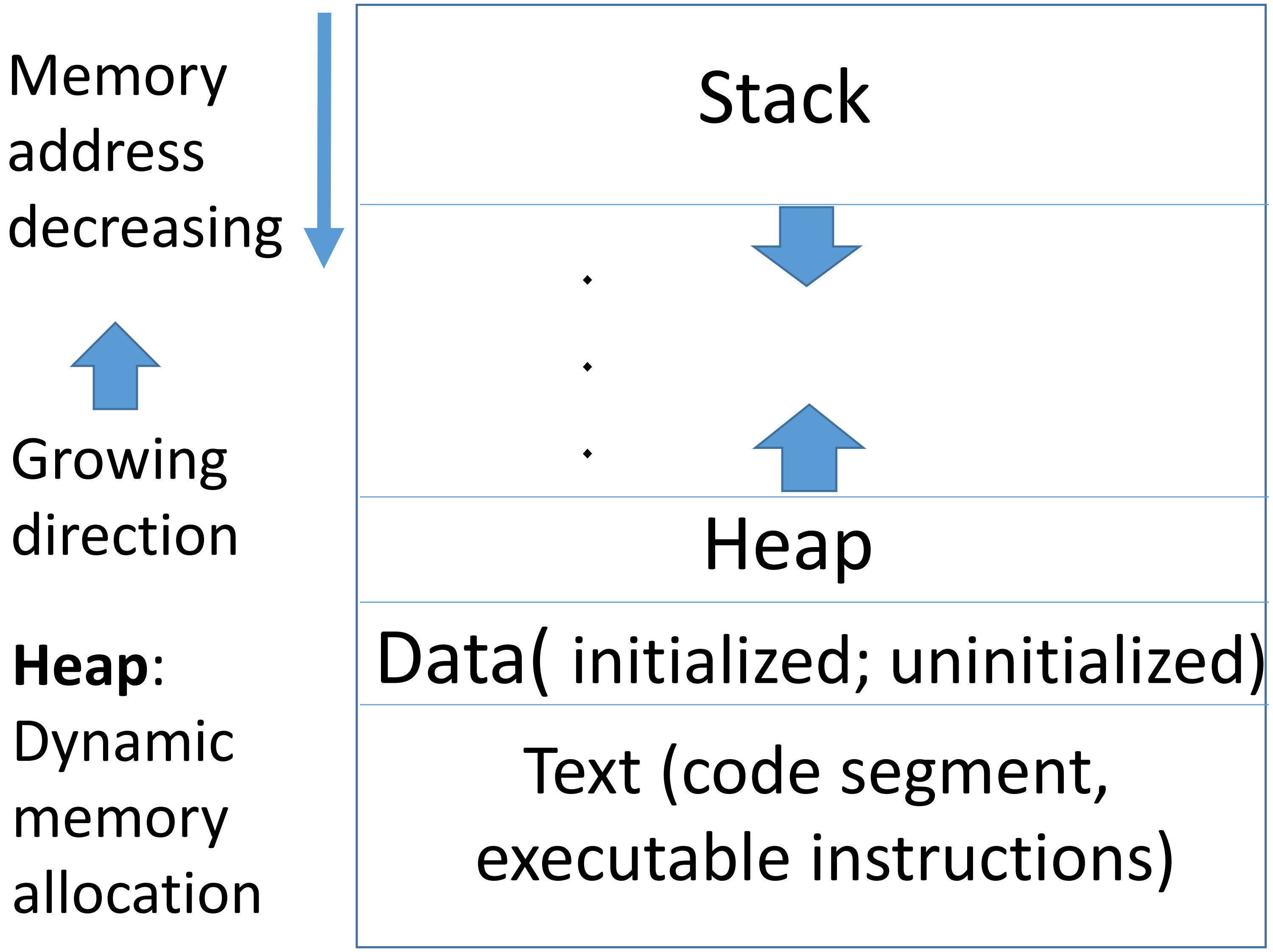
# Memory Layout

structure

# Memory Layout in C++



# Memory Layout in C++



Memory address decreasing

Growing direction

Stack

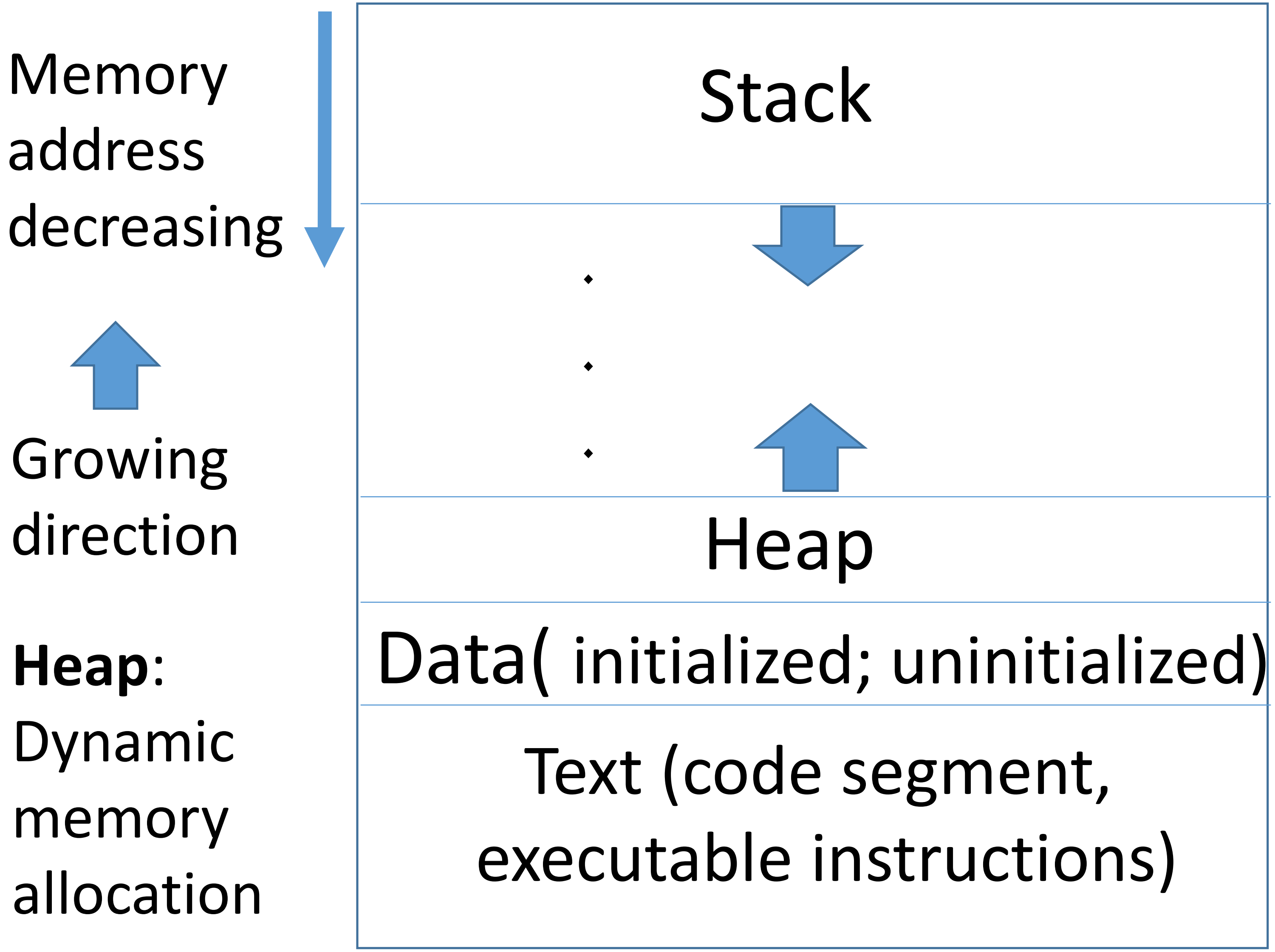
Heap

Data( initialized; uninitialized)

Text (code segment, executable instructions)

**Heap**: Dynamic memory allocation

# Memory Layout in C++



Memory address decreasing

Growing direction

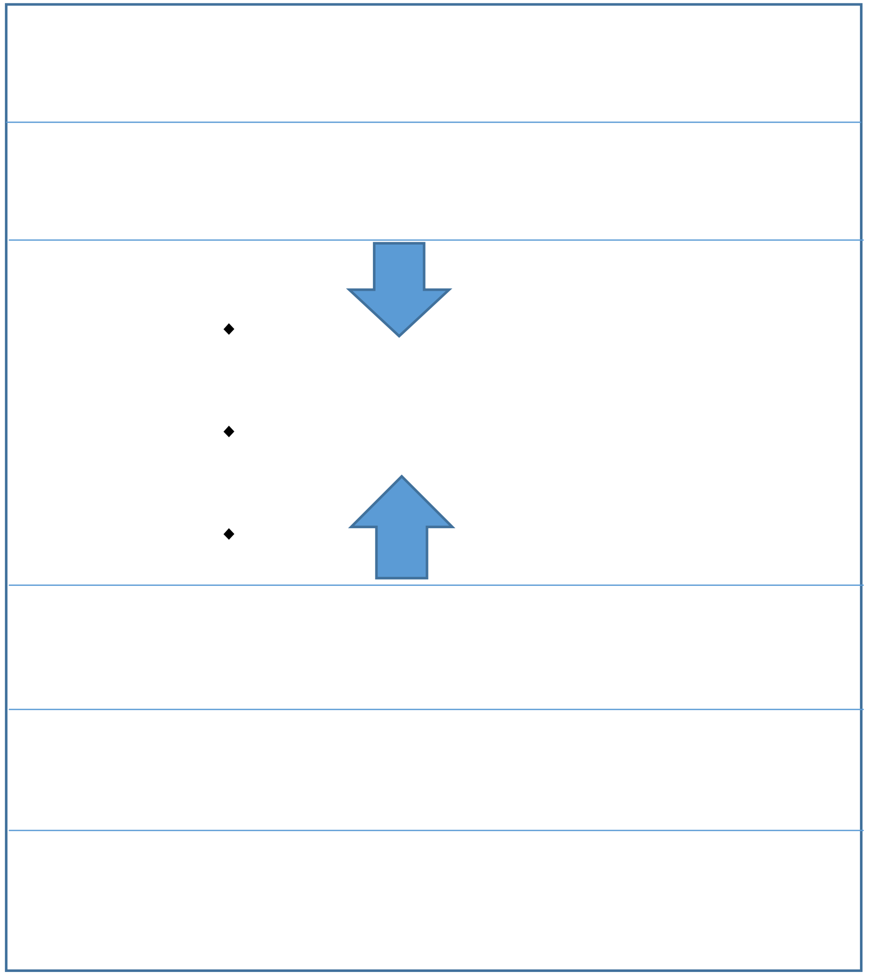
**Heap**: Dynamic memory allocation

Stack

Heap

Data( initialized; uninitialized)

Text (code segment, executable instructions)



An example

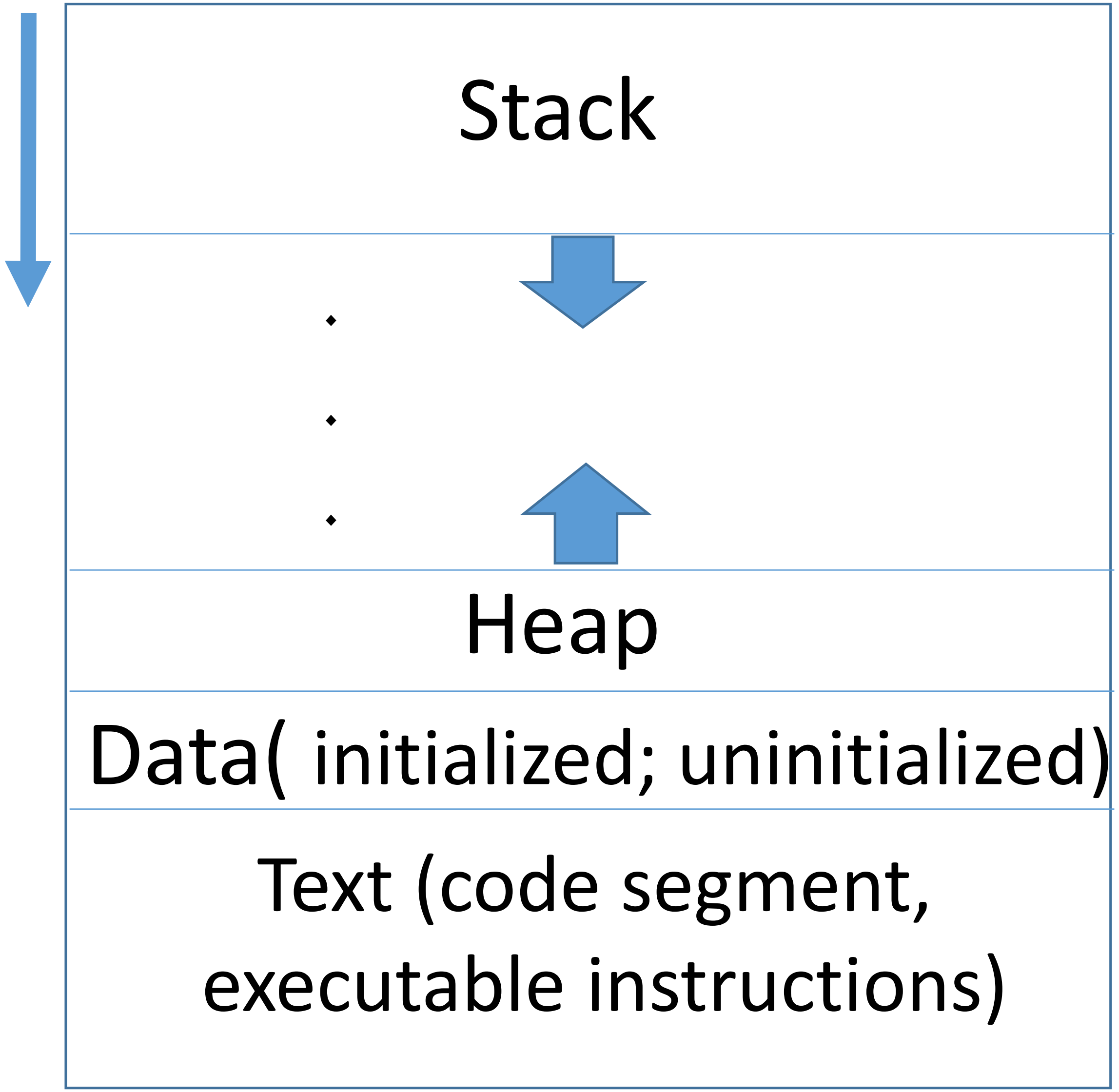
# Memory Layout in C++

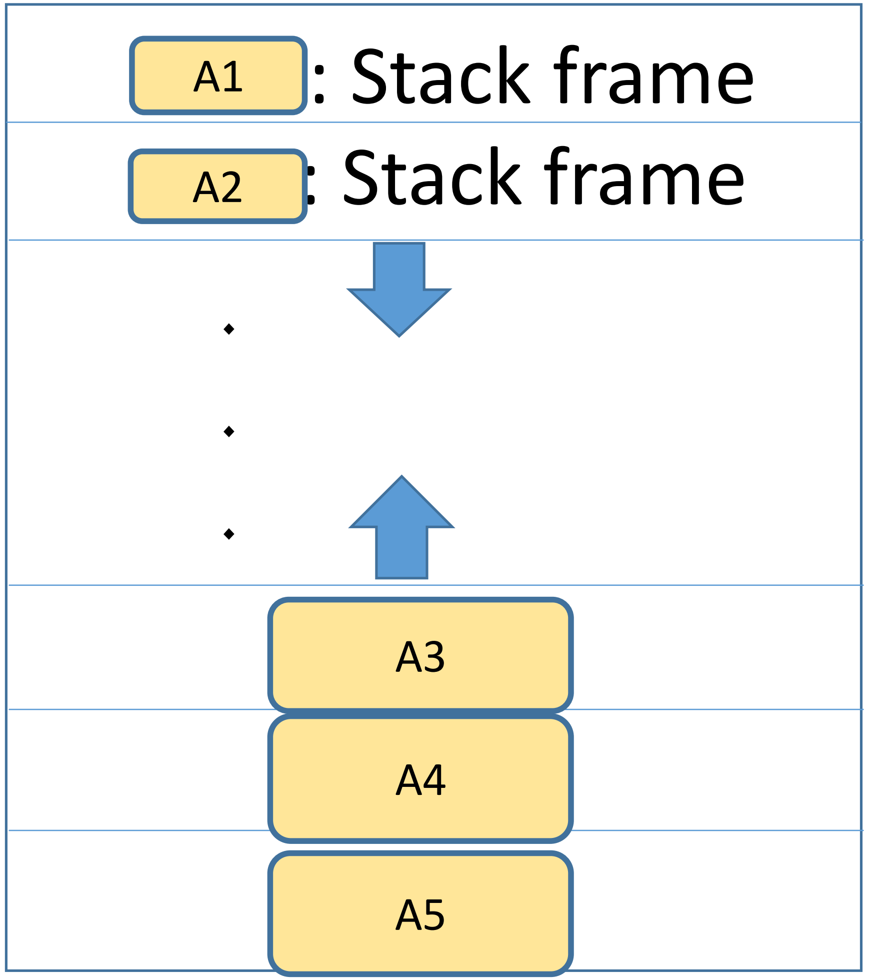
```
void main( … );
int max( …);
```

Memory address decreasing

Growing direction

**Heap**: Dynamic memory allocation


Stack
. . .
Heap
Data( initialized; uninitialized)
Text (code segment, executable instructions)



An example

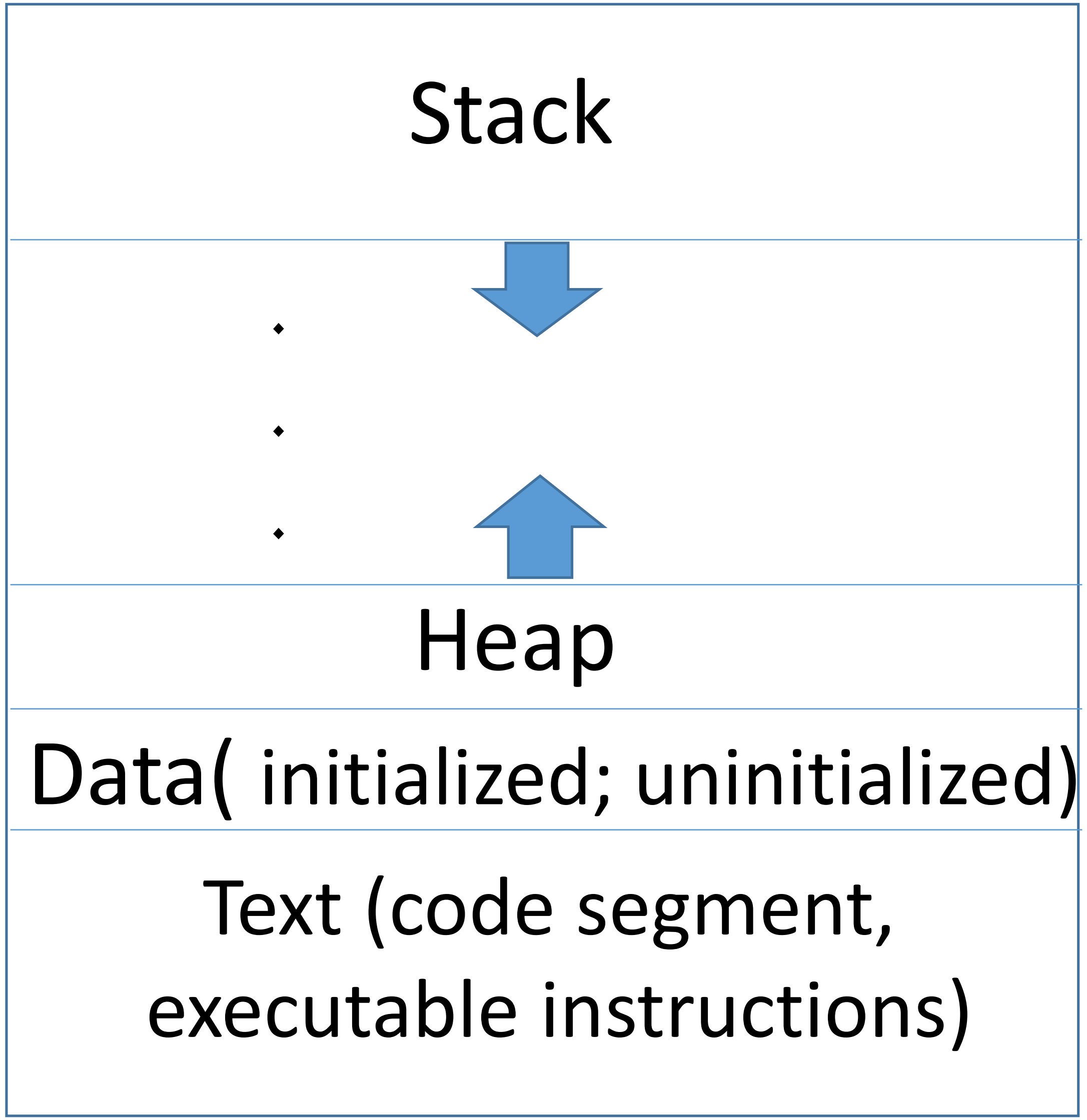
# Memory Layout in C++

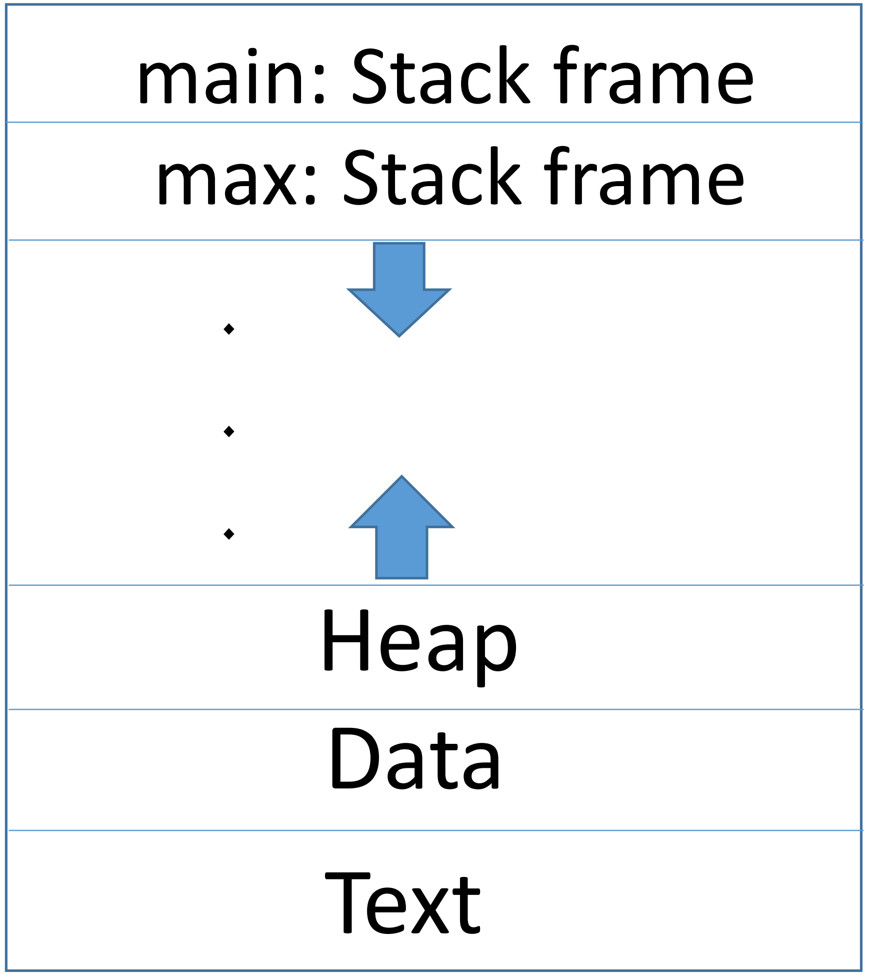
```
void main( … );
int max( …);
```

Memory address decreasing

Growing direction

**Heap**: Dynamic memory allocation


Stack
.
.
.
Heap
Data( initialized; uninitialized)
Text (code segment, executable instructions)



An example

# Function call

Work flow

```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```

```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```

```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```

## What happens when functions are invoked?

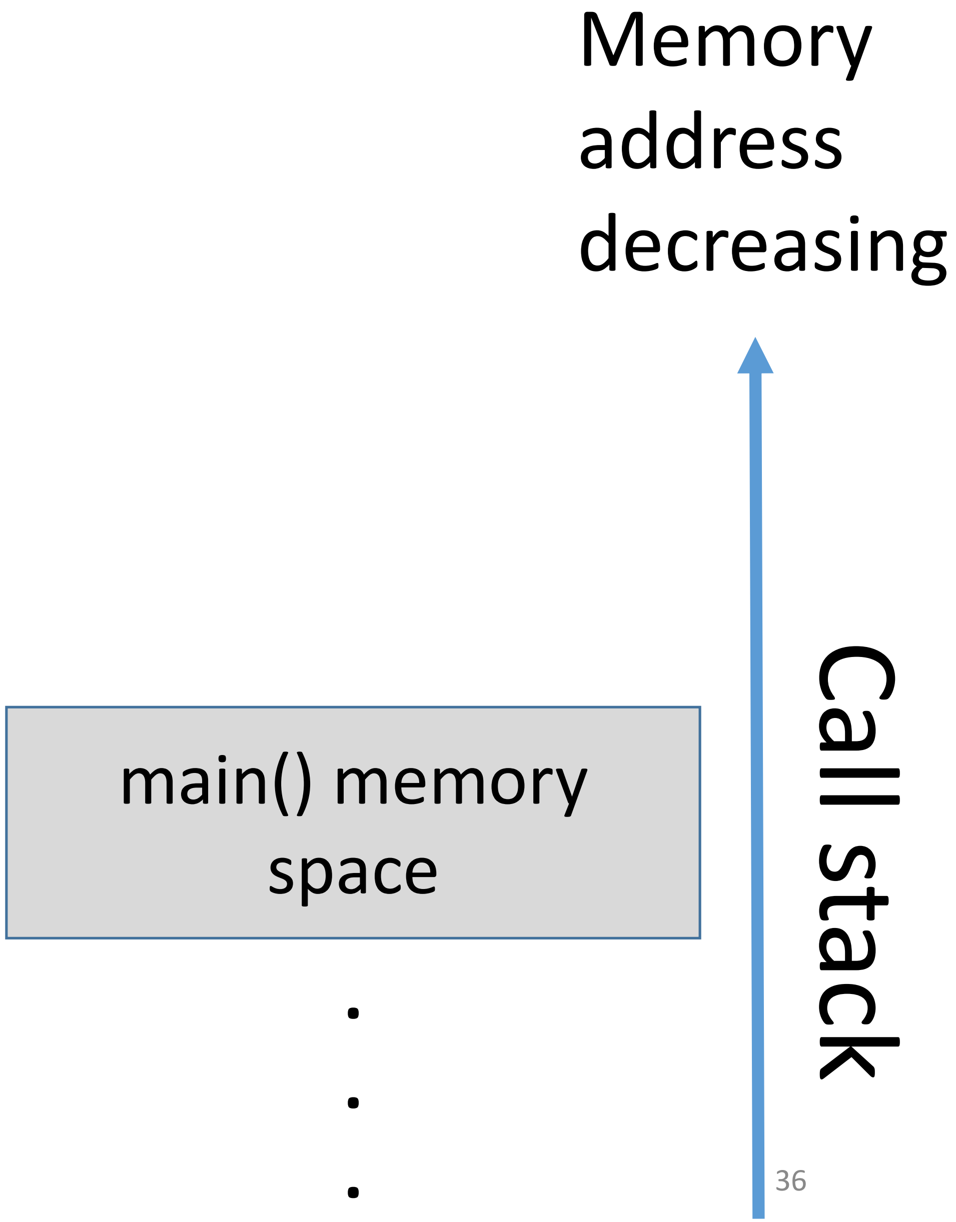
A A1 is used to store the information about the A2 functions (subroutines).

```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```

# Assume that main is called now.

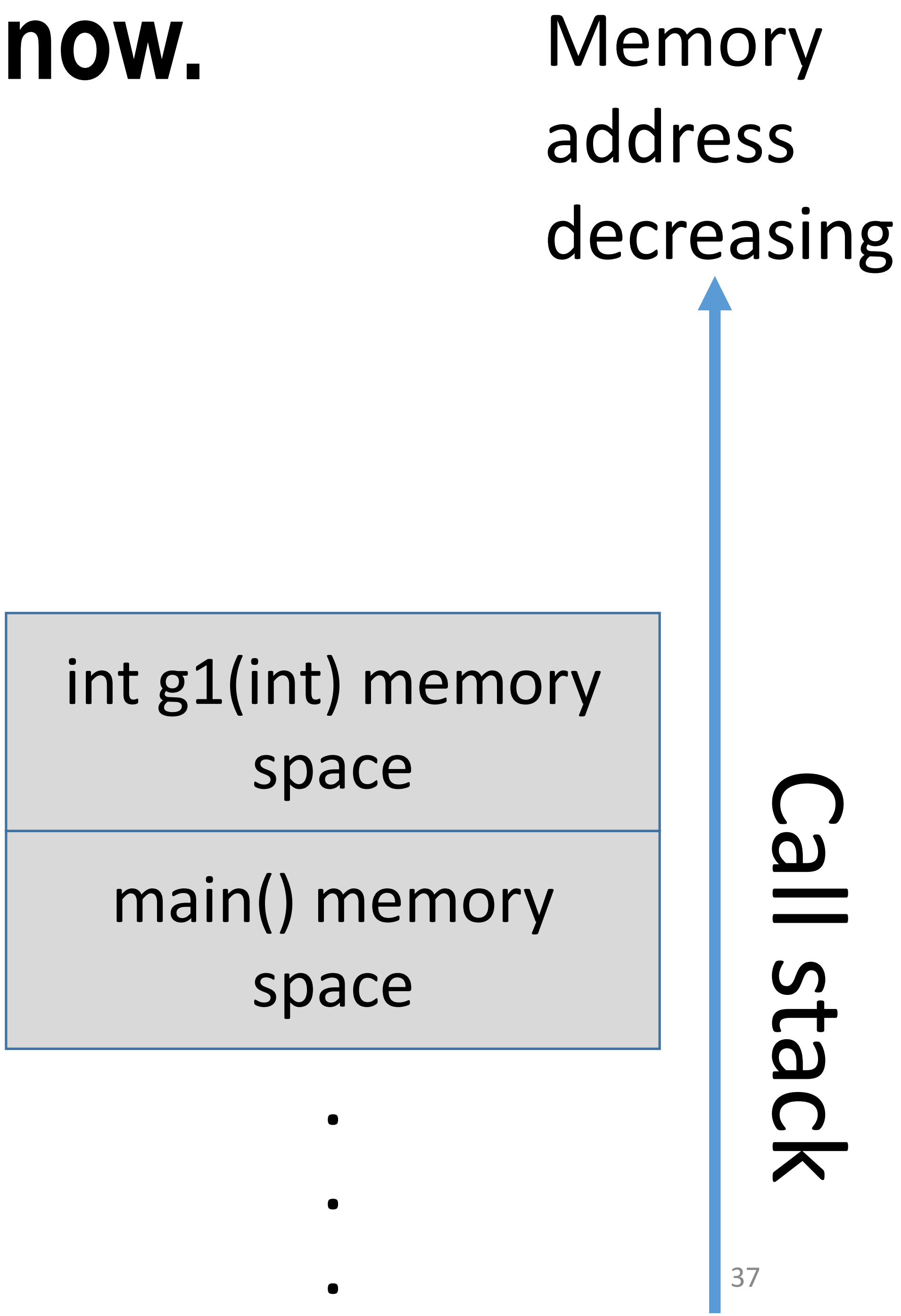
```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```

# Assume that main is called now.



# int g1(int) is called now.

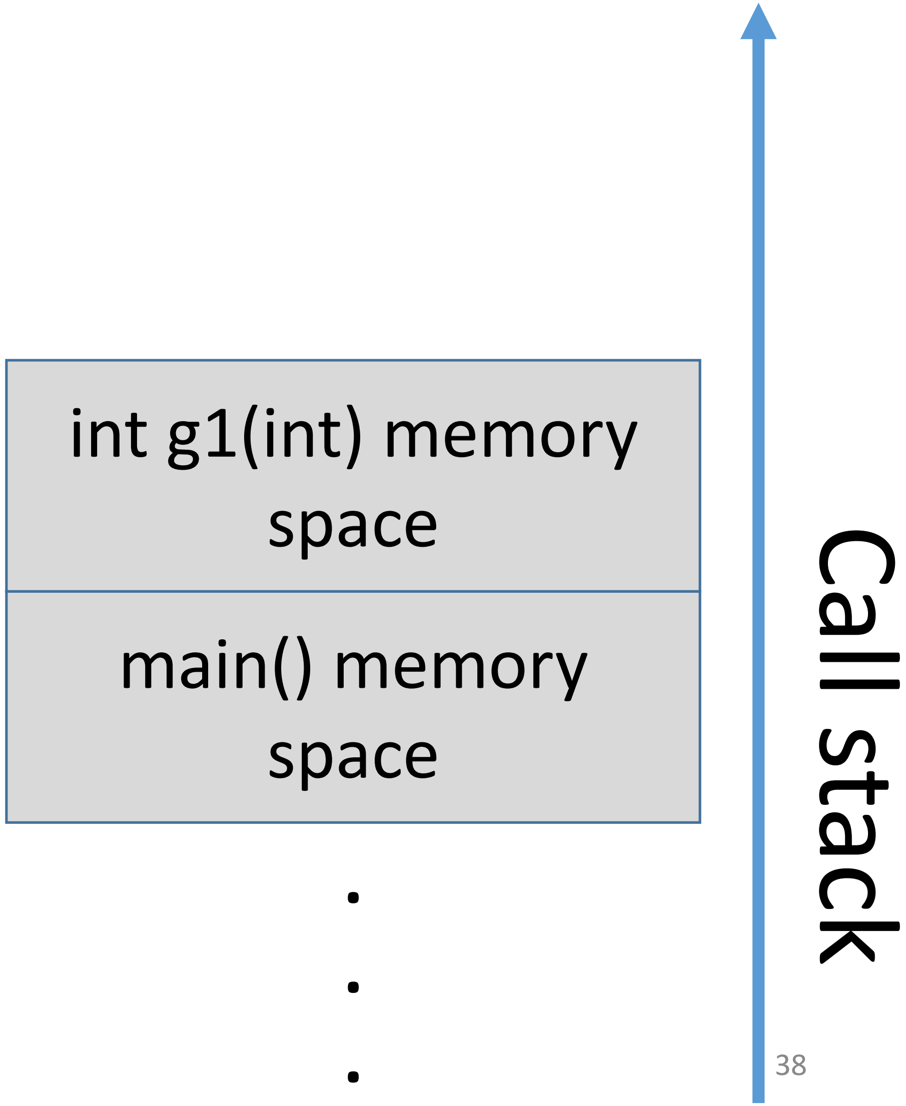
```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```



# int g1(int) is called now.

```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```

Memory address decreasing



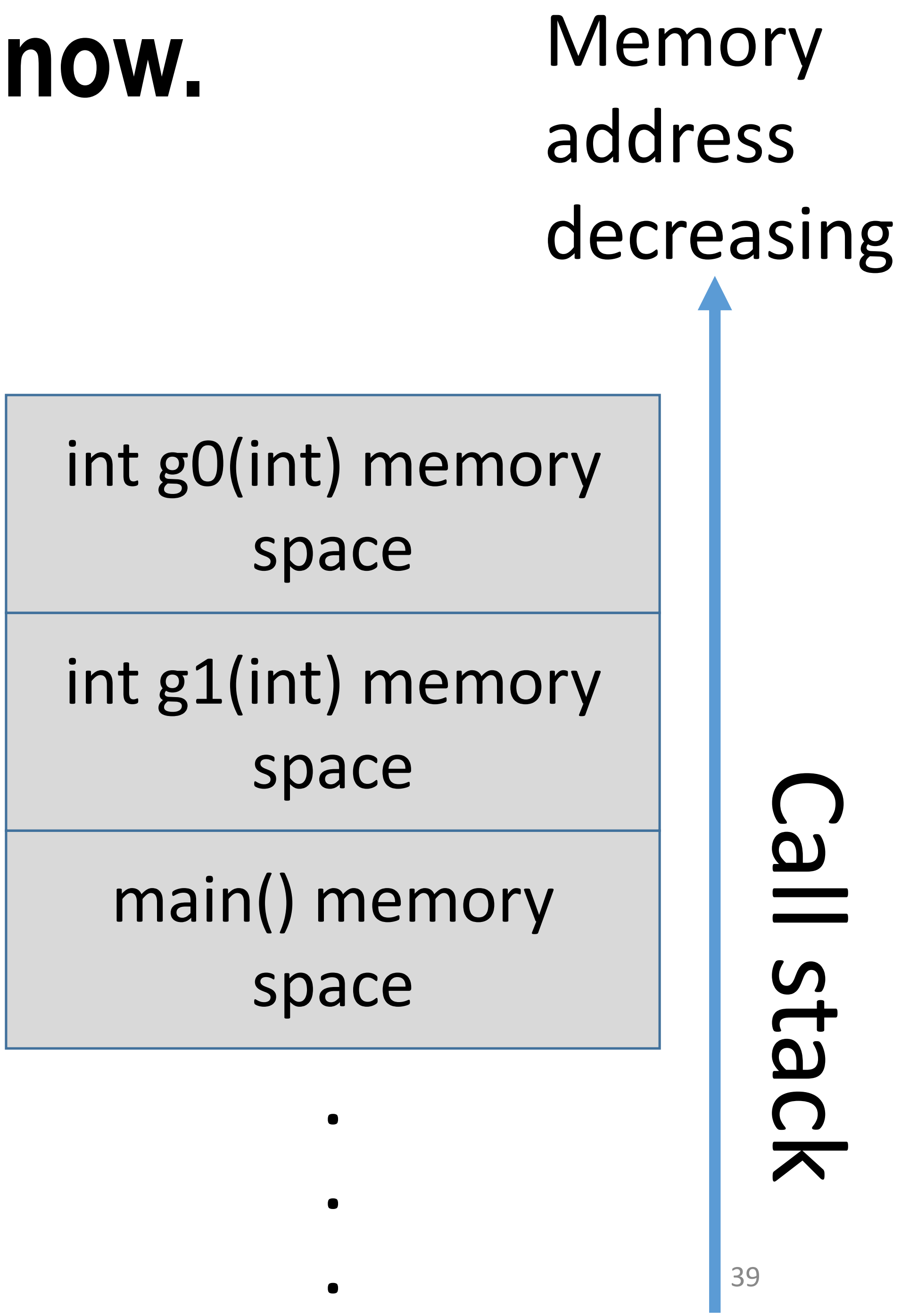
int g1(int) memory space

main() memory space

.
.
.

```
int g0(int k0) {
    int p0;
    ......
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ......
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ......
    p2 = g1(k2*k2);
    return p2;
}
```

# int g0(int) is called now.



Memory address decreasing

int g0(int) memory space

int g1(int) memory space

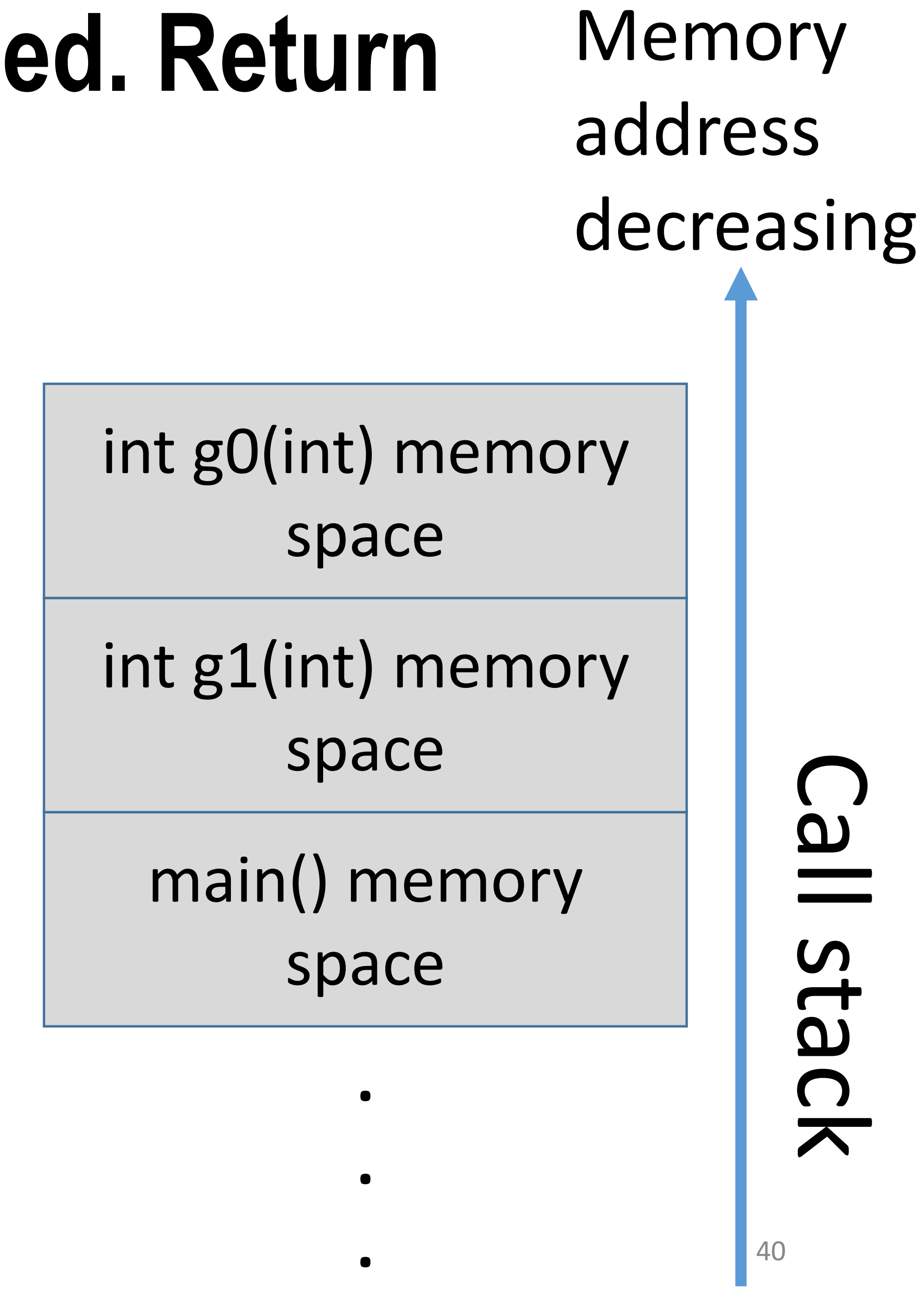
main() memory space

.
.
.

Call stack

```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```

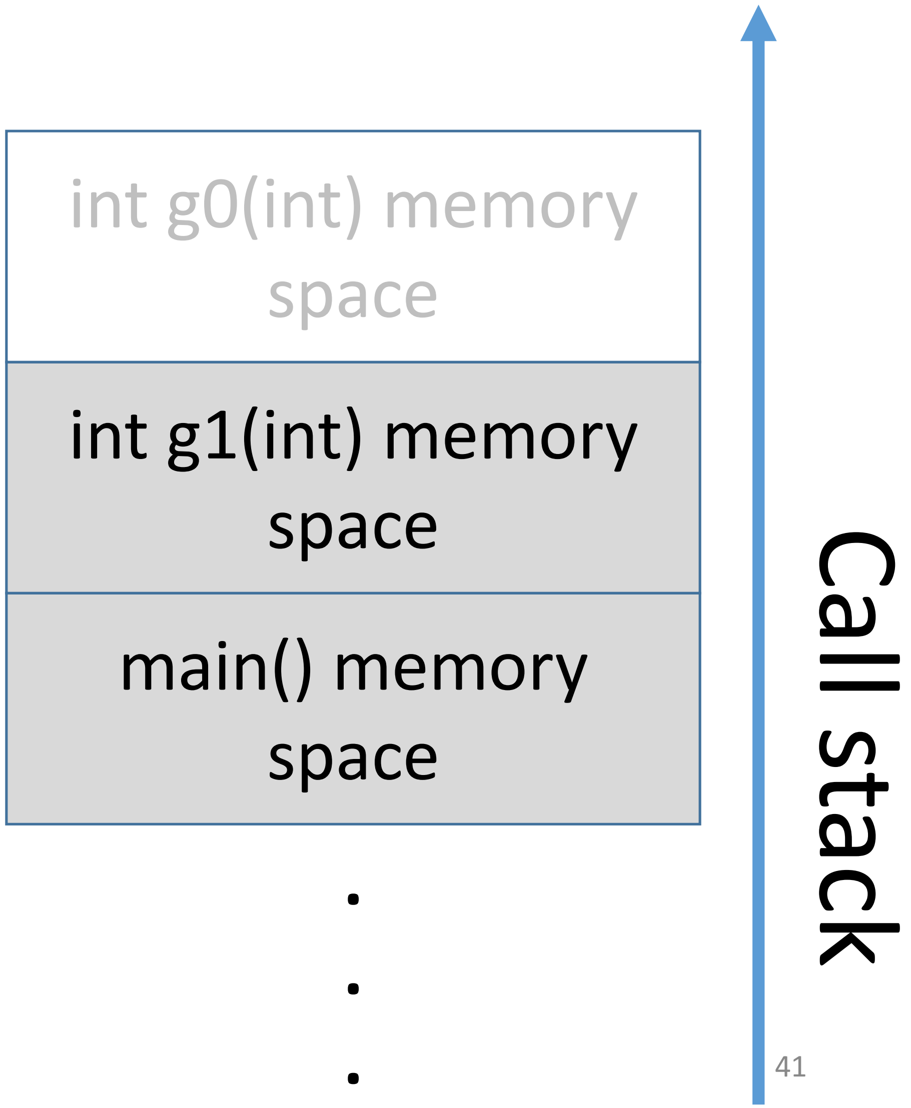
# int g0(int) is finished. Return to int g1(int).



```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```

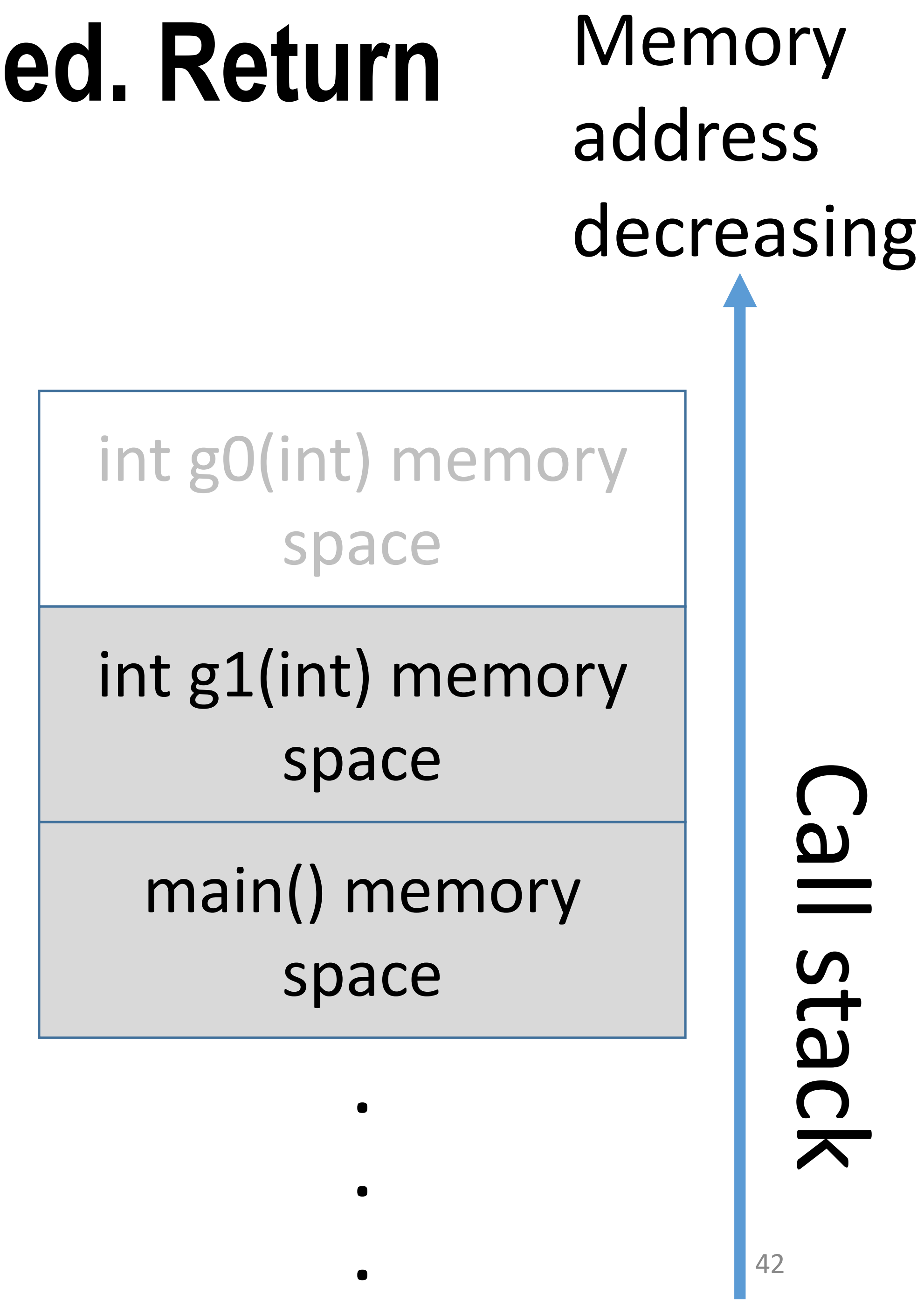
# int g0(int) is finished. Return to int g1(int).

Memory address decreasing



# int g1(int) is finished. Return to int main().

```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```



Memory address decreasing

int g0(int) memory space

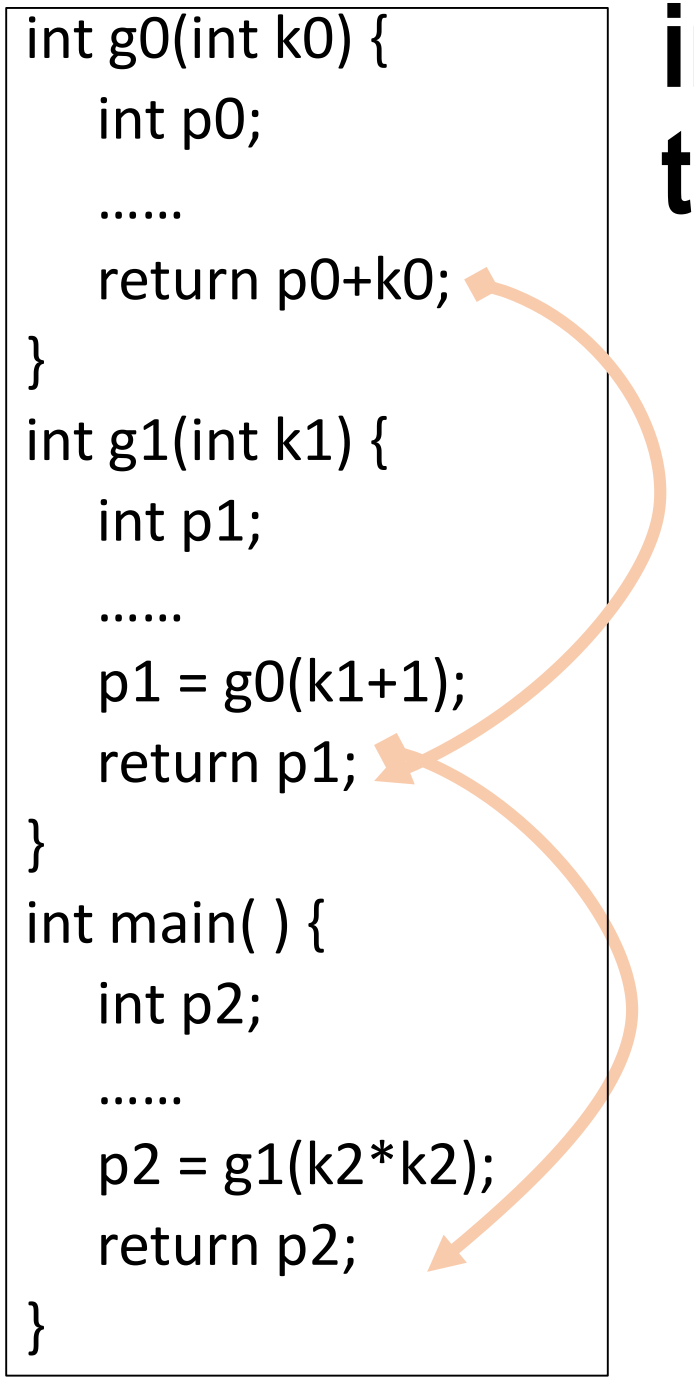
int g1(int) memory space

main() memory space

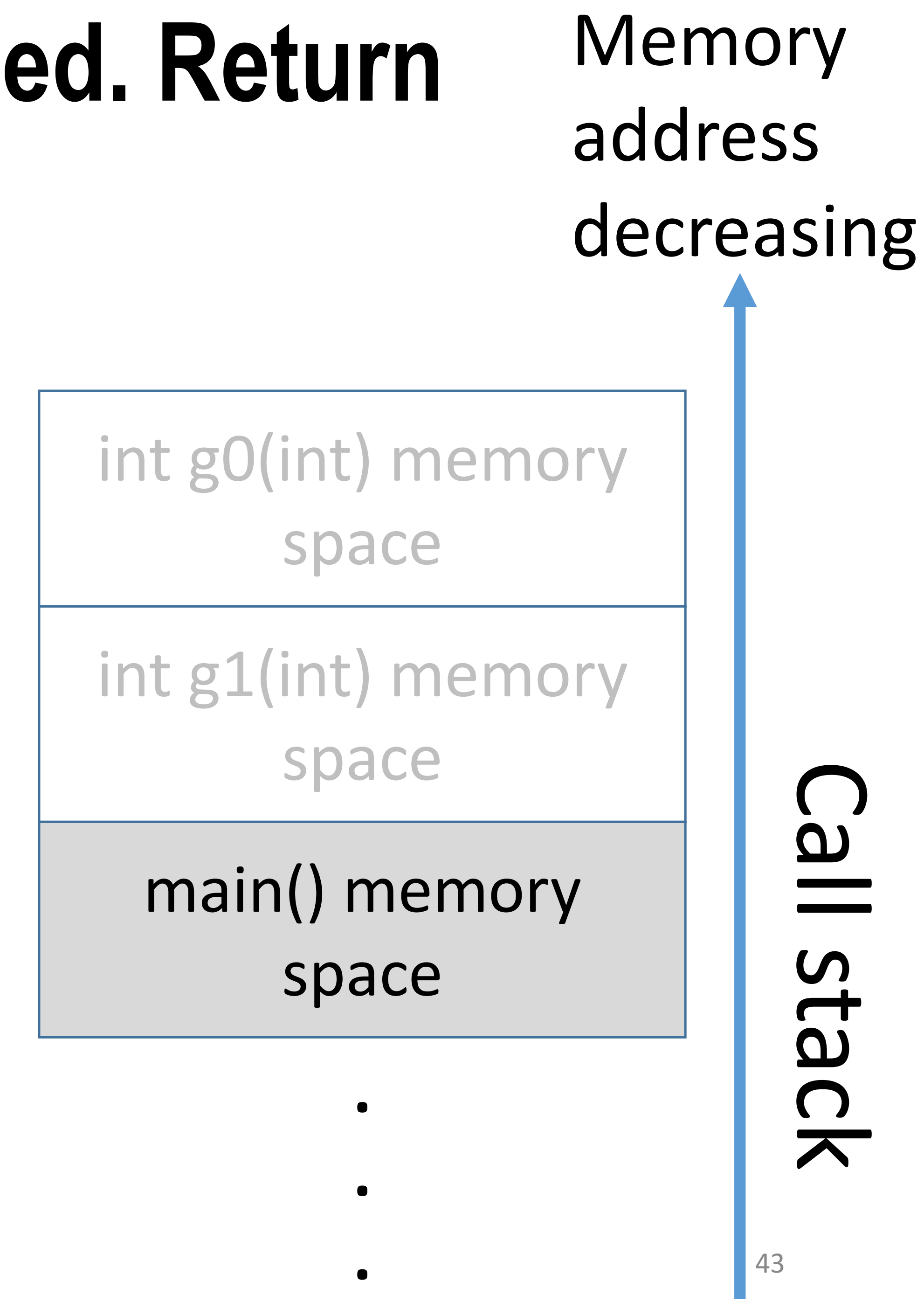
.
.
.

Call stack

# int g1(int) is finished. Return to int main().



```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```



Memory address decreasing

int g0(int) memory space

int g1(int) memory space

main() memory space

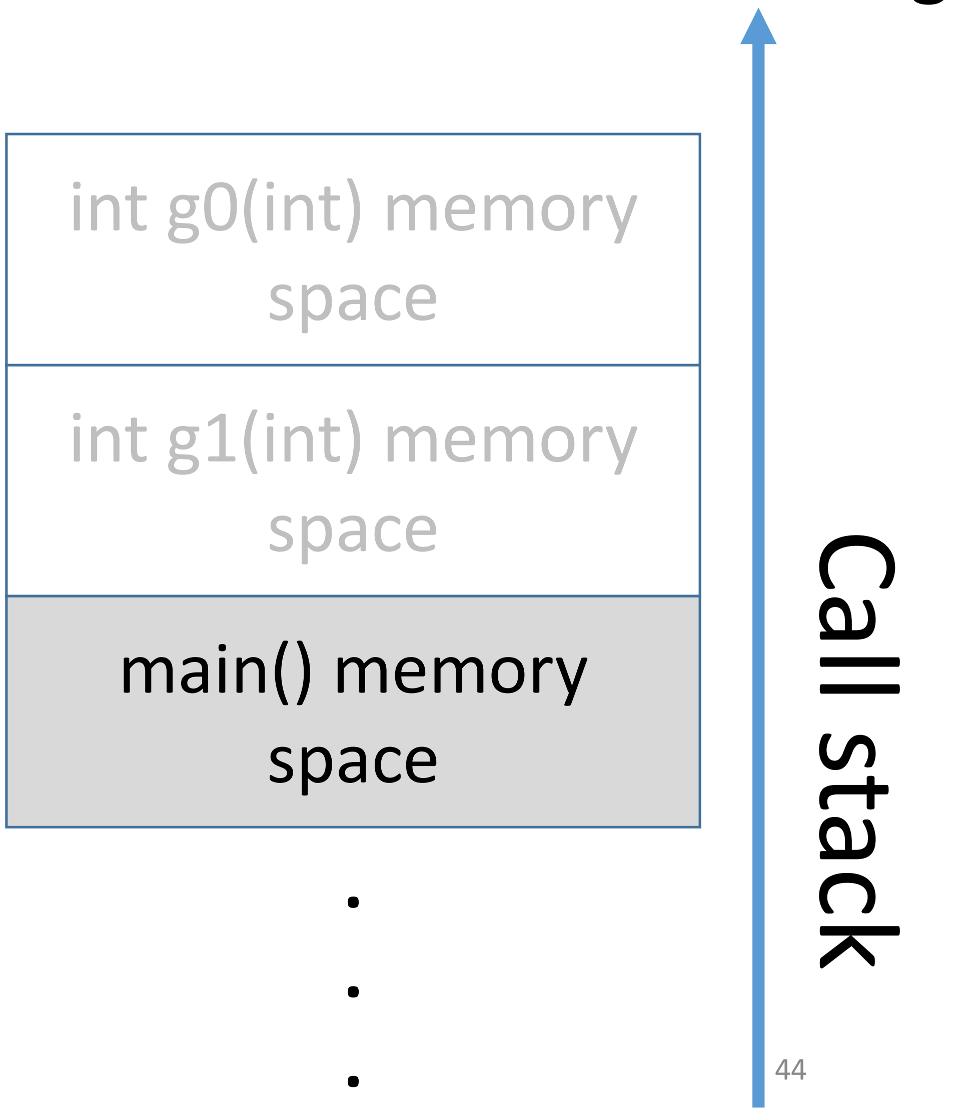
.
.
.

Call stack

# int main() is finished. Return to the caller.

```
int g0(int k0) {
    int p0;
    ......
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ......
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ......
    p2 = g1(k2*k2);
    return p2;
}
```

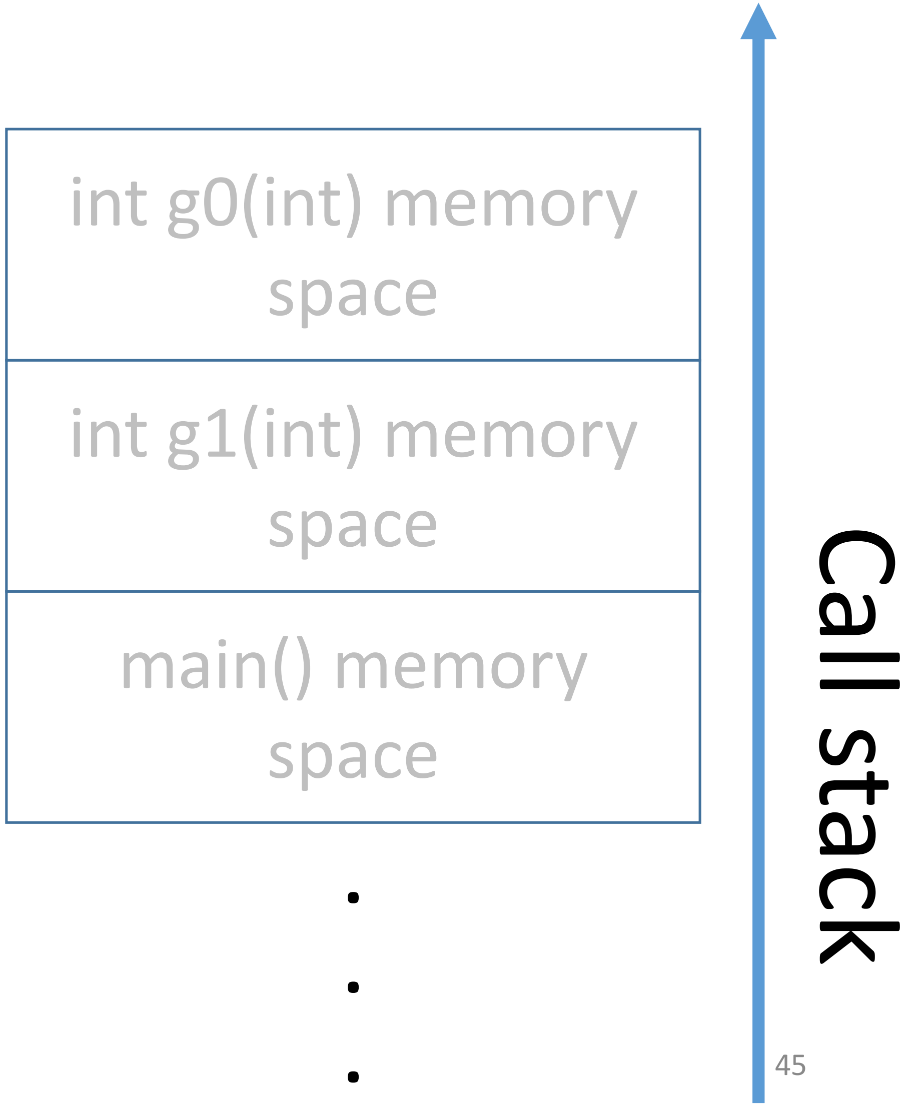
Memory address decreasing



```
int g0(int k0) {
    int p0;
    ……
    return p0+k0;
}
int g1(int k1) {
    int p1;
    ……
    p1 = g0(k1+1);
    return p1;
}
int main( ) {
    int p2;
    ……
    p2 = g1(k2*k2);
    return p2;
}
```

# int main() is finished. Return to the caller.

Memory address decreasing



int g0(int) memory space

int g1(int) memory space

main() memory space

.
.
.

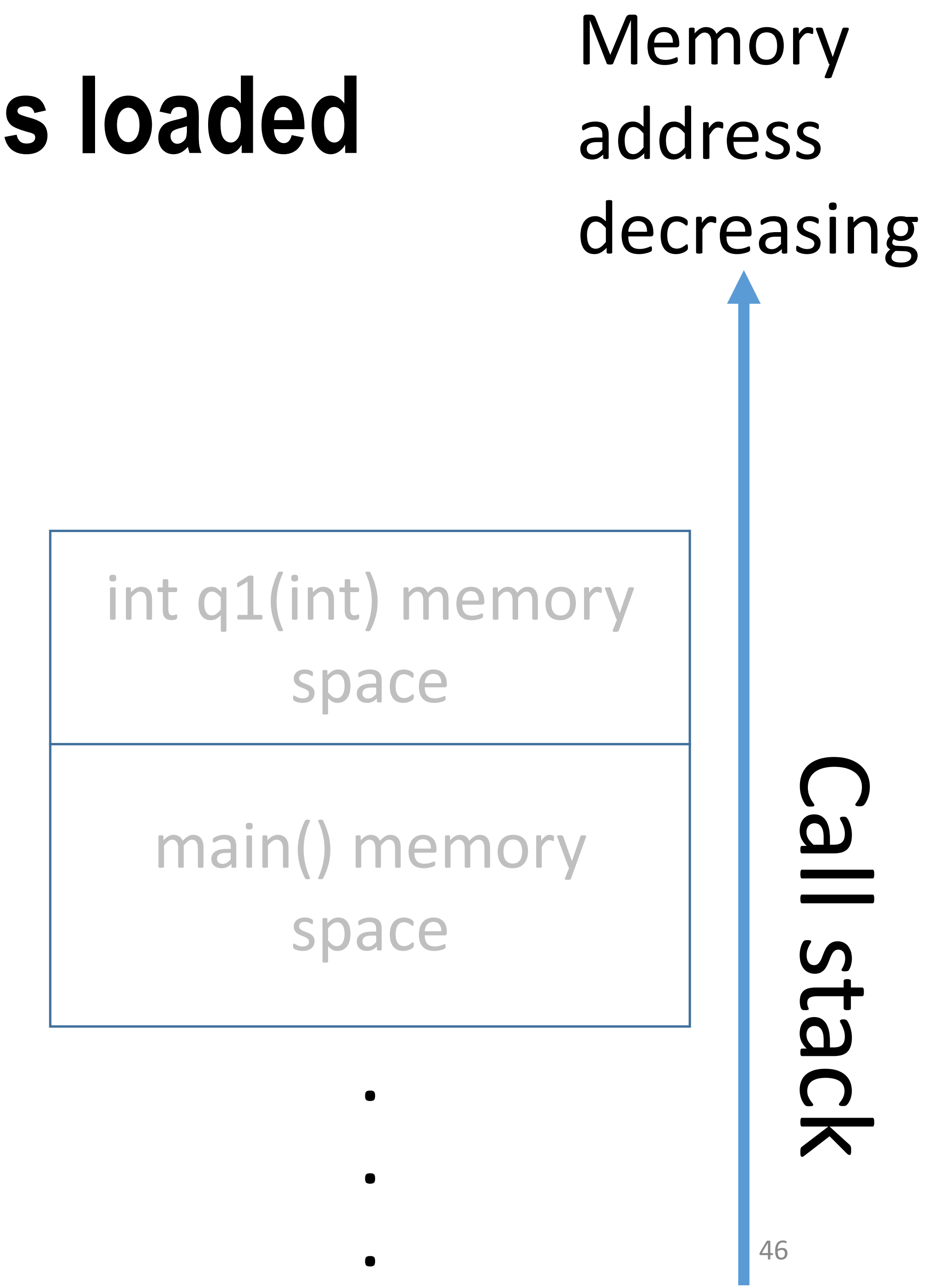
Call stack

# Another program is loaded

Another program is loaded to the same memory space.

If there are **uninitialized** local variables of functions, **their values** are **arbitrary**.

```
int q1(int k) {
    int p;
    ……
    return p;
}
int main( ) {
    int t = 4, u = 0;
    ……
    u = q1( t );
    return p2;
}
```



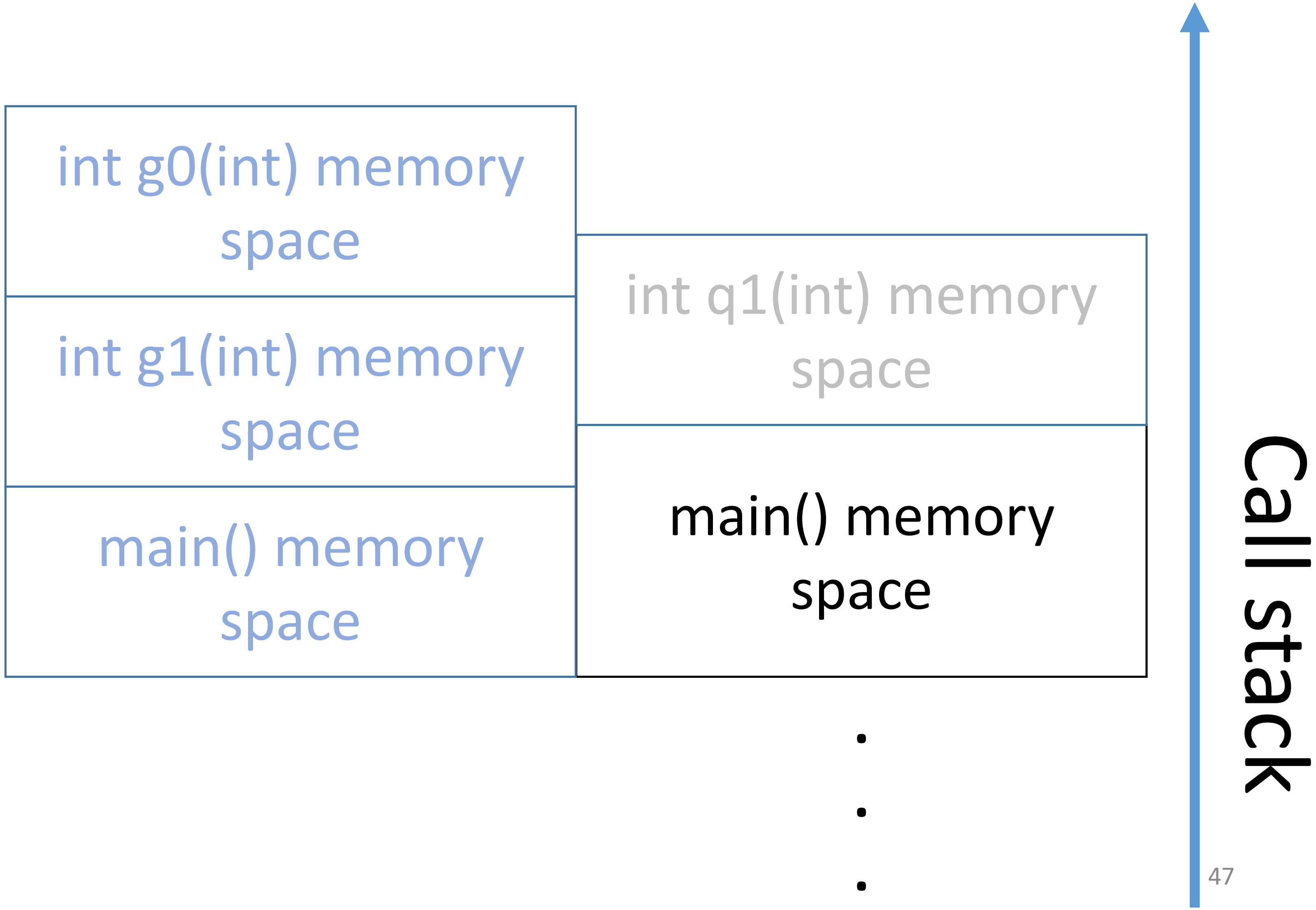
Another program is loaded to the same memory space.

If there are **uninitialized** local variables of functions, **their values** are **arbitrary**.

```
int q1(int k) {
    int p;
    ……
    return p;
}
int main( ) {
    int t = 4, u = 0;
    ……
    u = q1( t );
    return p2;
}
```

# Another program is loaded

Memory address decreasing

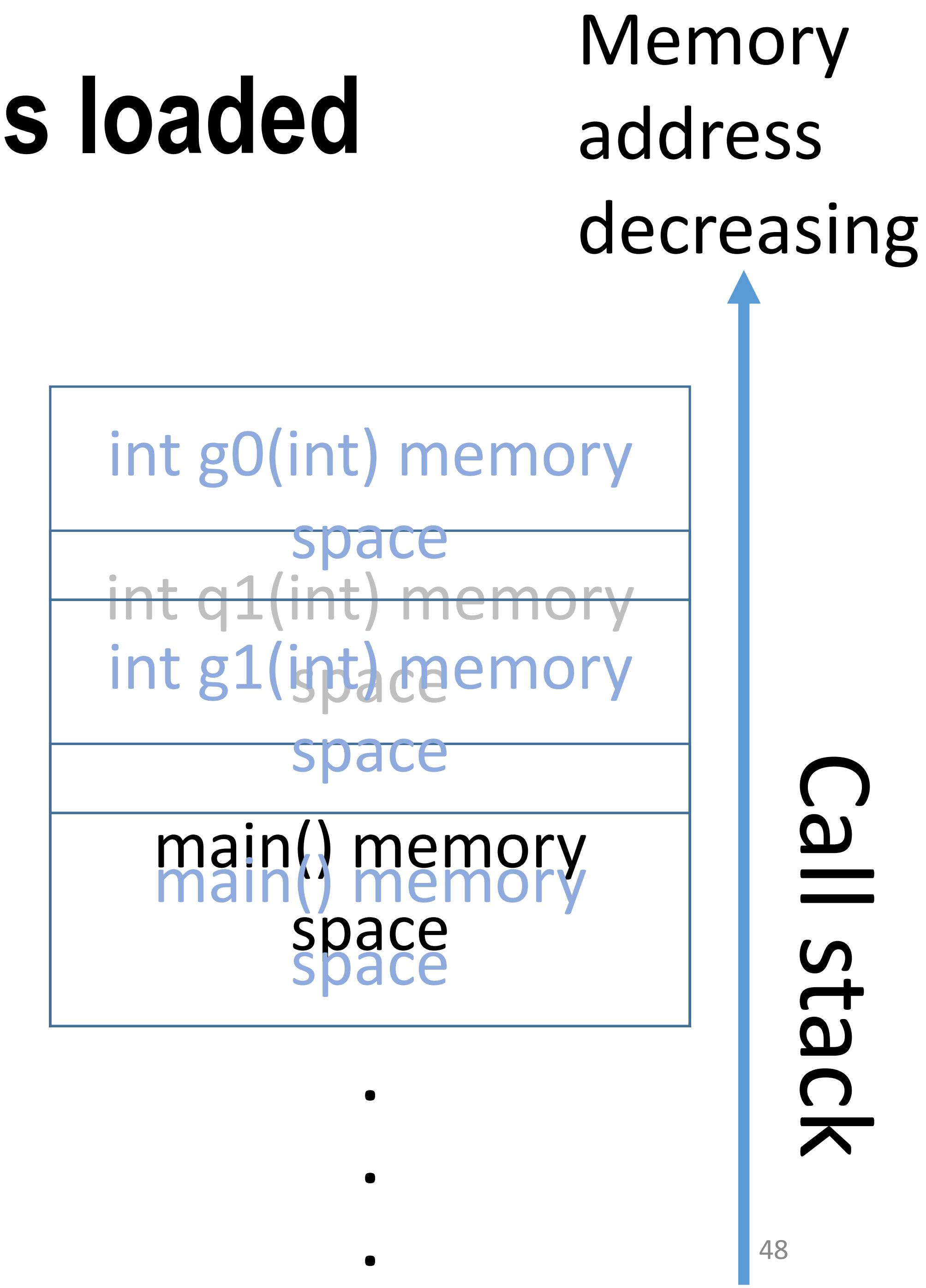


Another program is loaded to the same memory space.

If there are **uninitialized** local variables of functions, **their values** are **arbitrary**.

```
int q1(int k) {
    int p;
    ……
    return p;
}
int main( ) {
    int t = 4, u = 0;
    ……
    u = q1( t );
    return p2;
}
```

# Another program is loaded



# Another program is loaded

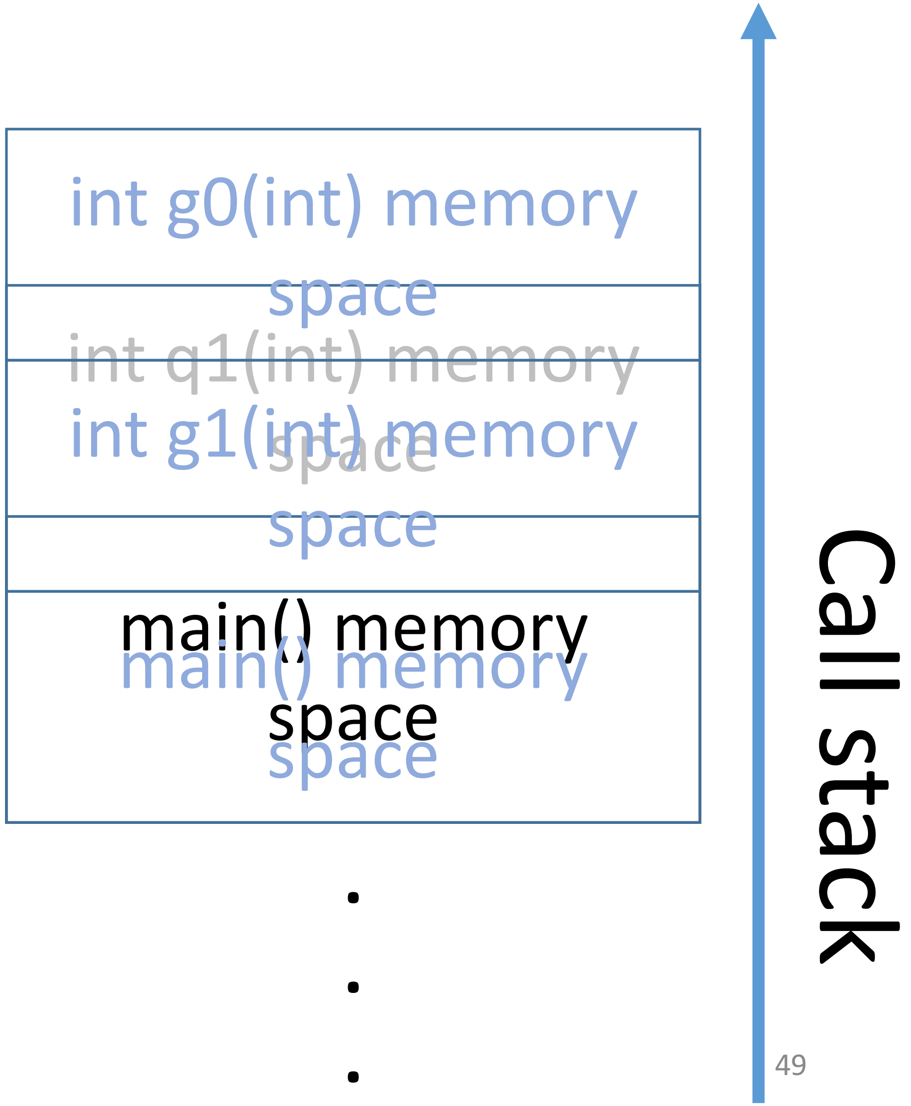
Another program is loaded to the same memory space.

If there are **uninitialized** local variables of functions, **their values** are **arbitrary**.

```
int q1(int k) {
    int p;
    ……
    return p;
}
int main( ) {
    int t = 4, u = 0;
    ……
    u = q1( t );
    return p2;
}
```

p’s value is arbitrary

Memory address decreasing



# Calling Functions

```
int main ( ) {
   int a = 15;
   int b = -12;

   int x = max (a, b);

   cout << “maximum between  “
      << a << “ and “ << b << “ is “
     << x << endl;

    return 0;
}
```

# Calling Functions

*parameter order association*

```
int main ( ) {
   int a = 15;
   int b = -12;

   int x = max (a, b);

   cout << “maximum between  “
      << a << “ and “ << b << “ is “
     << x << endl;

    return 0;
}
```

# Calling Functions

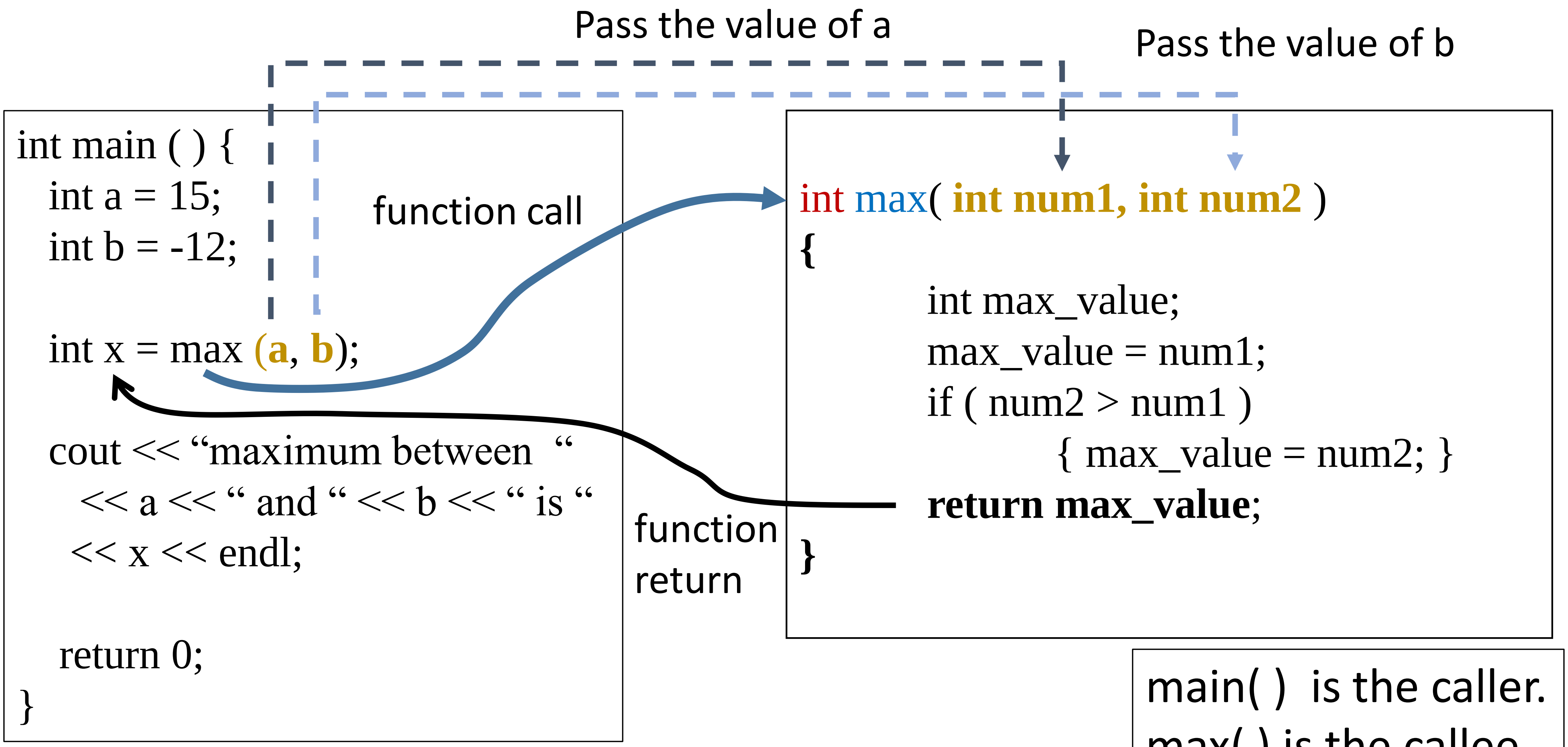
*parameter order association*

```
int main ( ) {
   int a = 15;
   int b = -12;

   int x = max (a, b);

   cout << “maximum between  “
      << a << “ and “ << b << “ is “
     << x << endl;

    return 0;
}
```

```
int max( int num1, int num2 )
{
            int max_value;
            max_value = num1;
            if ( num2 > num1 )
                        { max_value = num2; }
            return max_value;
}
```

# Calling Functions

*parameter order association*



Pass the value of a

Pass the value of b

```
int main ( ) {
   int a = 15;
   int b = -12;

   int x = max (a, b);

   cout << “maximum between  “
     << a << “ and “ << b << “ is “
    << x << endl;

    return 0;
}
```

function call

function return

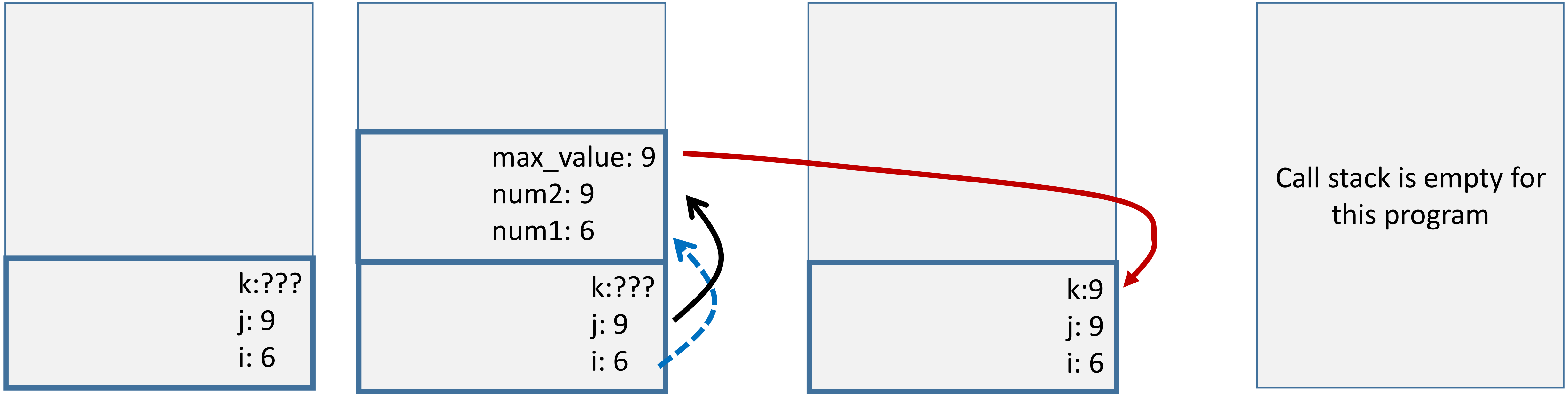
```
int max( int num1, int num2 )
{
          int max_value;
          max_value = num1;
          if ( num2 > num1 )
                    { max_value = num2; }
          return max_value;
}
```

main( )  is the caller.
max( ) is the callee.

# Call Stacks

```
int main() {
  int i = 6;
  int j = 9;
  int k = max(i, j);
  cout << "The maximum value is: "
       << k << endl;
  return 0;
}
```

```
int max( int num1, int num2 )
{
        int max_value;
        max_value = num1;
        if ( num2 > num1 )
                { max_value = num2; }
        return max_value;
}
```



1. Invoke the main function.

2. Invoke the max function and execute it.

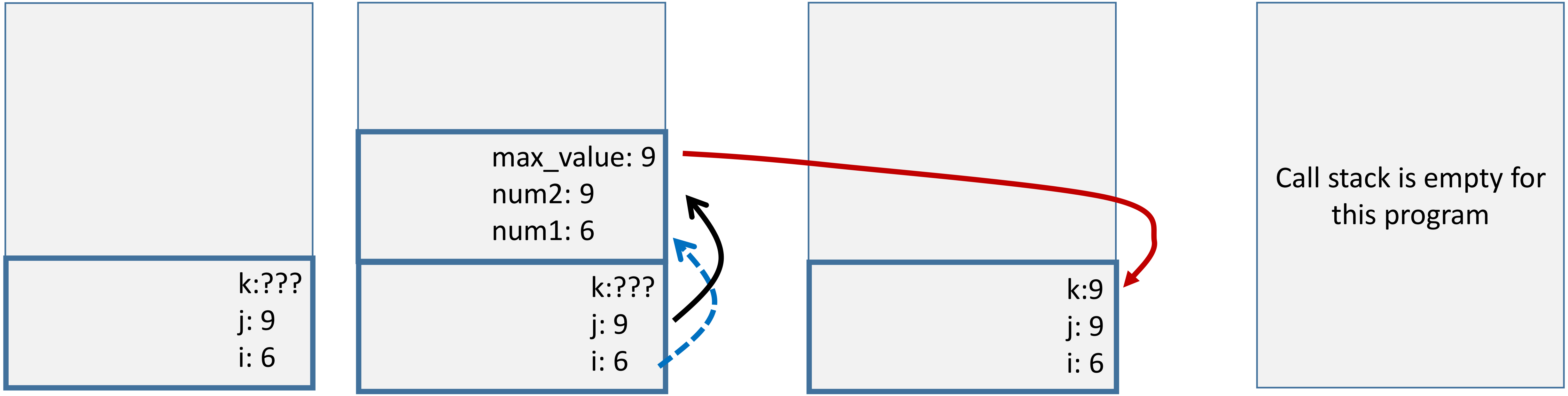
3. Function max is finished. The value of max_value is passed to k. Return to main.

4. main is finished.

# Call Stacks

```
int main() {
  int i = 6;
  int j = 9;
  int k = max(i, j);
  cout << "The maximum value is: "
       << k << endl;
  return 0;
}
```

```
int max( int num1, int num2 )
{
        int max_value;
        max_value = num1;
        if ( num2 > num1 )
                { max_value = num2; }
        return max_value;
}
```



k:???
j: 9
i: 6

max_value: 9
num2: 9
num1: 6

k:???
j: 9
i: 6

k:9
j: 9
i: 6

Call stack is empty for this program

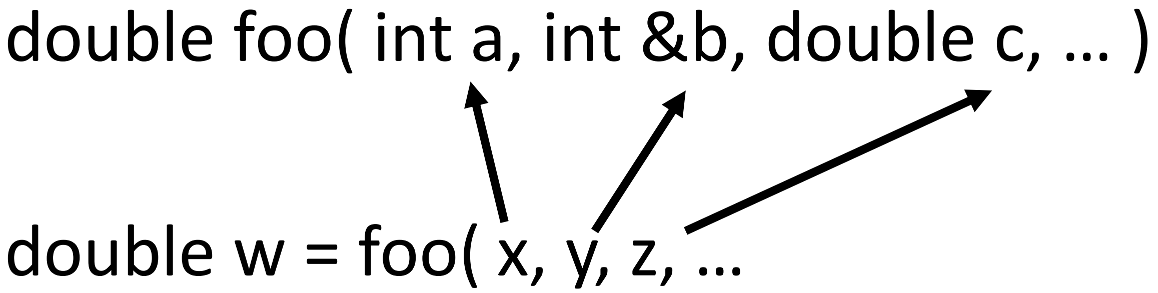
1. Invoke the main function.

2. Invoke the max function and execute it.

3. Function max is finished. The value of max_value is passed to k. Return to main.

4. main is finished.

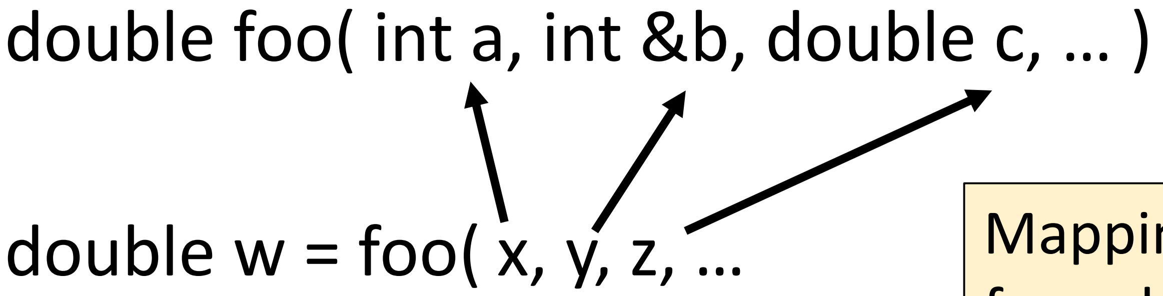
# Parameter order association



double foo( int a, int &b, double c, … )

double w = foo( x, y, z, …

# Parameter order association



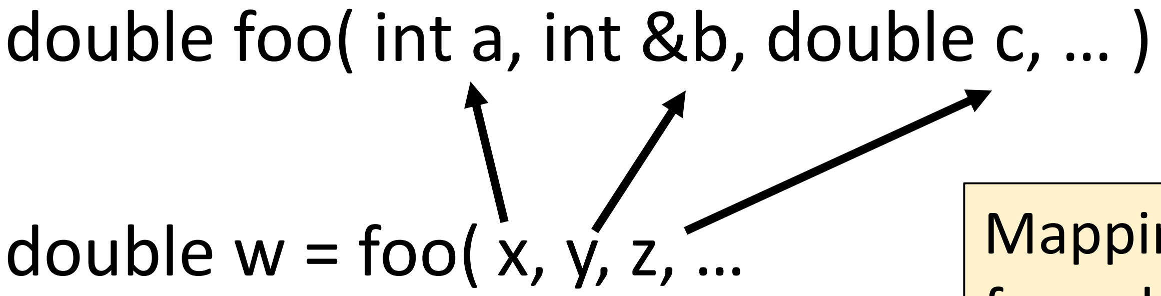
double foo( int a, int &b, double c, … )

double w = foo( x, y, z, …

Mapping the actual parameters to the formal parameters

# Parameter order association

- ➢ When calling a function, we need to provide A1

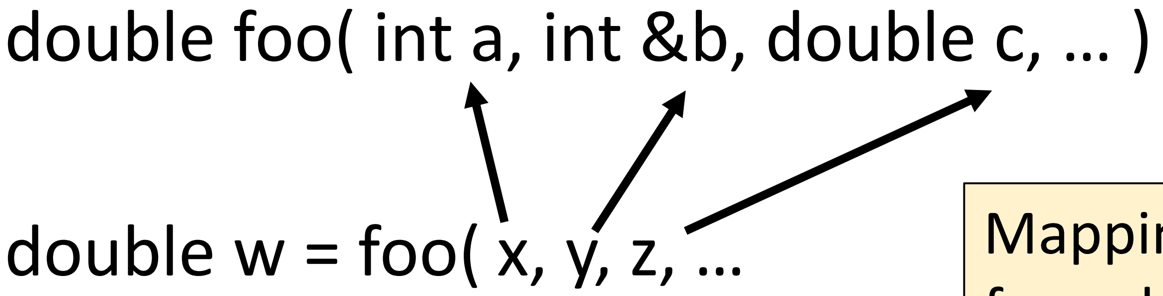


```
double foo( int a, int &b, double c, … )

double w = foo( x, y, z, …
```

Mapping the actual parameters to the formal parameters

# Parameter order association

- When calling a function, we need to provide arguments.
- The arguments must A1 in the A2 order as their respective parameters in the function A3



double foo( int a, int &b, double c, … )

double w = foo( x, y, z, …

Mapping the actual parameters to the formal parameters

# Overloading Functions

Functions with the A1 name but they have parameters of A2 s.

**int min(int, int)** **double min(double, double)**

# Ambiguous Invocation

```
int maxNumber(int num1, double num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

```
double maxNumber(double num1, int num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

```
#include <iostream>
using namespace std;
……
int main()  {
  cout << maxNumber( 2, 3.0) << endl;    // error?
  cout << maxNumber(1, 2) << endl;        // error?
  return 0;
}
```

# Ambiguous Invocation

```
int maxNumber(int num1, double num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

```
double maxNumber(double num1, int num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

```
#include <iostream>
using namespace std;
……
int main()  {           int  double
  cout << maxNumber( 2, 3.0) << endl;    // error?
  cout << maxNumber(1, 2) << endl;        // error?
  return 0;             int  int
}
```

# Ambiguous Invocation

```
int maxNumber(int num1, double num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

```
double maxNumber(double num1, int num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

```
#include <iostream>

using namespace std;

......

int main()  {            int  double

  cout << maxNumber( 2, 3.0) << endl;    // A1. error?

  cout << maxNumber(1, 2) << endl;       // A2. error?

  return 0;              int  int

}
```

# Ambiguous Invocation

```
int maxNumber(int num1, double num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

```
double maxNumber(double num1, int num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

```
#include <iostream>
using namespace std;
……
int main()  {
  cout << maxNumber( 2, 3.0) << endl;    // ok
  cout << maxNumber(1, 2) << endl;        // error
  return 0;
}
```

It is a A1. Type of error? error.

Sometimes there may be A2 **possible matches** for an invocation of a function.

The compiler A3 determine the most specific match.

# Ambiguous Invocation

```
int maxNumber(int num1, double num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

```
double maxNumber(double num1, int num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

```
#include <iostream>
using namespace std;
……
int main()  {
  cout << maxNumber( 2, 3.0) << endl;    // ok
  cout << maxNumber(1, 2) << endl;        // error
  return 0;
}
```

It is a compilation error.

Sometimes there may be **two or more possible matches** for an invocation of a function.

The compiler **cannot** determine the most specific match.

# Function Prototypes

```

void g( ) {
        int x = foo(10, 12);
        ……
}
int foo( int a, int b) {
   return a*b;
}
```

Not good

```
int foo( int, int ); // ???
void g( ) {
        int x = foo(10, 12);
        ……
}
int foo( int a, int b) {
   return a*b;
}
```

☺

# Function Prototypes

```
void g( ) {
            int x = foo(10, 12);
            ……
}
int foo( int a, int b) {
    return a*b;
}
```

Not good

```
int foo( int, int ); //
void g( ) {
            int x = foo(10, 12);
            ……
}
int foo( int a, int b) {
    return a*b;
}
```

A1

# Function Prototypes

A function must be declared before it is called.

Two proper ways:

1. Place the declaration before all function calls.

2. Declare a function prototype before the function is called.

A function prototype is a function declaration without implementation (no body).

```
void g( ) {
        int x = foo(10, 12);
        ……
}
int foo( int a, int b) {
   return a*b;
}
```

error

```
int foo( int, int ); // forward declaration
void g( ) {
        int x = foo(10, 12);
        ……
}
int foo( int a, int b) {
   return a*b;
}
```

# Default Arguments

```
void k(char ch = ‘a’) {
        cout << “k():\t” << ch << endl;
}
class A {
        public:
        void f( int y = 10) {
                cout << “A   y\t” << y << endl;
        }
};
```

```
void main( ) {
        k( b );
        k( );
        A x;
        x.f( );
}
```

```
k():        b
k():        a
A   y       10
```

# Default Arguments

A function with default argument values has A1 assignment expressions.

The default values are passed to the parameters when the function is invoked A2

```
void k(char ch = ‘a’) {
        cout << “k():\t” << ch << endl;
}
class A {
        public:
        void f( int y = 10) {
                cout << “A   y\t” << y << endl;
        }
};
```

```
void main( ) {
        k( b );
        k( );
        A x;
        x.f( );
}
```

```
k():        b
k():        a
A   y       10
```

# Default Arguments

A function with default argument values has one or more assignment expressions.

The default values are passed to the parameters when the function is invoked without the arguments.

The default parameters must be defined A1 all the A2 parameters

```
void k(int c, char ch = ‘a’, string str = “good”) {// ok
      cout << “k():” << ch << endl;
}

void f( char ch = ‘a’, int k, string str = “good”) {// error
      cout << “f():” << ch << endl;
}
```

# Default Arguments

A function with default argument values has one or more assignment expressions.

The default values are passed to the parameters when the function is invoked without the arguments.

The default parameters must be defined A1 all the A2 parameters

```
void k(int c, char ch = ‘a’, string str = “good”) {// ok
      cout << “k():” << ch << endl;
}

void f( char ch = ‘a’, int k, string str = “good”) {// error
      cout << “f():” << ch << endl;
}
```

```
k(10);
f(‘b’)
```

# Scope of Variables

```
{
   int b; // block enclosing b and nearest to b
} the (nearest)
```

# Scope of Variables

```
{
   int b; // block enclosing b and nearest to b
} the (nearest)
```

A block is enclosed by a pair of braces { and }.

# Scope of Variables

- ➢A local variable: a variable defined A1 a function.
- ➢Scope: the part of the program where the variable A2 .
- ➢The scope of a variable starts from A3 and continues to the A4 that contains the variable.

```
{
   int b; // block enclosing b and nearest to b
} the (nearest)
```

A block is enclosed by a pair of braces { and }.

# Scope of Variables

- ➢A local variable: a variable defined inside a function.
- ➢Scope: the part of the program where the variable can be referenced.
- ➢The scope of a variable starts from its declaration and continues to the end of the block that contains the variable.

```
{
   int b;
   …
}
b = 6;
```

{

   int b; // block enclosing b and **nearest** to b

} the (nearest)

A block is enclosed by a pair of braces { and }.

# Scope of Local Variables

- We can declare a local variable with the same name in different blocks.
- These variables are not the same.

```
void f( ) {
   int a;                        // L1
}
```

# Scope of Local Variables

- We can declare a local variable with the same name in different blocks.
- These variables are not the same.

```
void f( ) {
   int a;                          // L1
}

void g( ) {
   int a;                          // L2
   {
        int a;                     // L3
   }
}
```

# Scope of Local Variables

- We can declare a local variable with the same name in different blocks.
- These variables are not the same.

```
void f( ) {
   int a;                  // L1
}

void g( ) {
   int a;                  // L2
   {
      int a;               // L3
   }
}
```

What is the scope of a?

# Scope of Local Variables

- We can declare a local variable with the same name in different blocks.
- These variables are not the same.

Function body

```
void f( ) {
   int a;                    // L1
}
```

Function body

Inner block

```
void g( ) {
   int a;                    // L2
   {
      int a;                 // L3
   }
}
```

What is the scope of a?

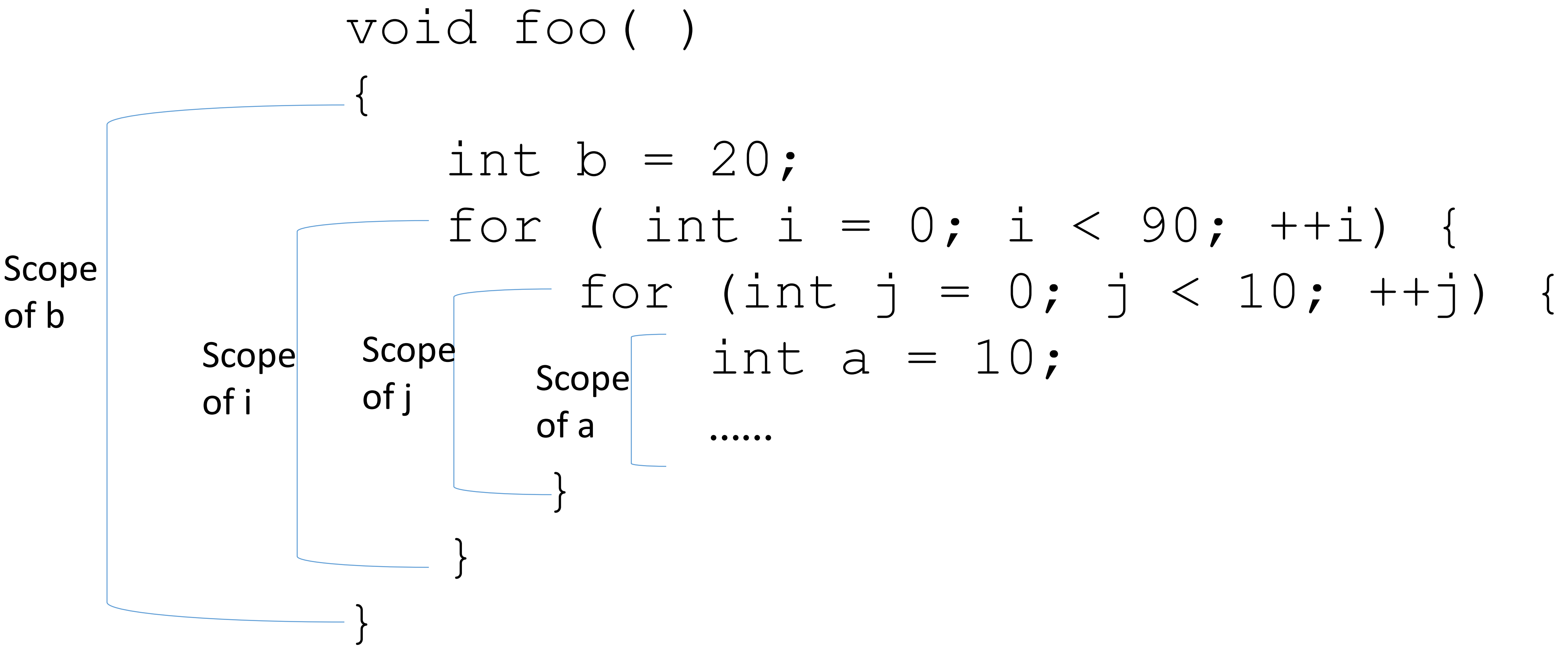
# Scope of Local Variables

What is the scope of each variable?

```
void foo( )
{
    int b = 20;
    for ( int i = 0; i < 90; ++i) {
        for (int j = 0; j < 10; ++j) {
            int a = 10;
            ……
        }
    }
}
```

# Scope of Local Variables

- ➢A variable in the initial action part of a for loop header: its scope is in the entire loop.
- ➢A variable declared inside a for loop body: its scope is in the loop body.



```
void foo( )
{
   int b = 20;
   for ( int i = 0; i < 90; ++i) {
       for (int j = 0; j < 10; ++j) {
           int a = 10;
           ......
      }
   }
}
```

Scope of b

Scope of i

Scope of j

Scope of a

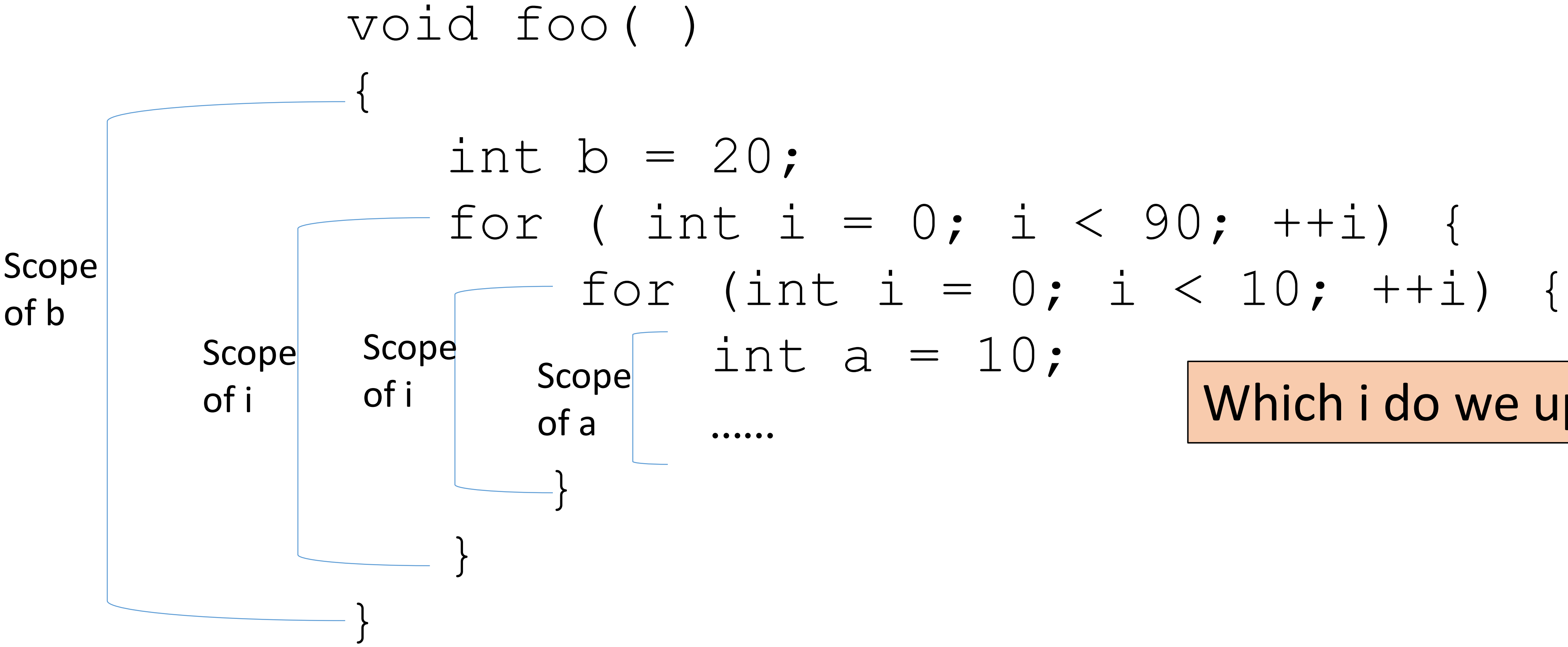
# Scope of Local Variables

Any problem?

```
void foo( )
{
   int b = 20;
   for ( int i = 0; i < 90; ++i) {
       for (int i = 0; i < 10; ++i) {
           int a = 10;
           ......
      }
   }
}
```

# Scope of Local Variables

Any problem?



```
void foo( )
{
    int b = 20;
    for ( int i = 0; i < 90; ++i) {
        for (int i = 0; i < 10; ++i) {
            int a = 10;
            ......
        }
    }
}
```

Which i do we update?

# Global Variables

- They are declared A1 all functions
- They are accessible to A2 in its scope.
- Global variables are defaulted to A3 .

# Global Variables

- They are declared outside all functions
- They are accessible to all functions in its scope.
- Global variables are defaulted to zero.

```
int a;    // global variable

class myClass {
   protected:  int myData;
   public:
   myClass( ) { myData = 100; }
   int getValue( ) const {
              return myData ;
   }
};

int main( )
{
              myClass myObj;
              a = myObj.getValue( );
              return 0;
}
```

# Global Variables

- They are declared outside all functions
- They are accessible to all functions in its scope.
- Global variables are defaulted to zero.

```
int a = 1;          // set a value to it

class myClass {
  protected:  int myData;
  public:
  myClass( ) { myData = 100; }
  int getValue( ) const {
            return myData ;
  }
};

int main( )
{
            myClass myObj;
            a = myObj.getValue( );
            return 0;
}
```

# Unary Scope Resolution

Unary scope resolution operator: **::**

```
#include <iostream>
using namespace std;
int y = 15;
int main()
{
  int y = 10;
  cout << "local variable x1 is " << y << endl;
  cout << "global variable x1 is " << ::y << endl;
  return 0;
}
```

# Static Local Variables

```
void fooA( ) {
        int counter = 0;
        ++counter;
        cout << counter << endl;
}
```

local variables of the function are destroyed

# Static Local Variables

```
void fooA( ) {
        int counter = 0;
        ++counter;
        cout << counter << endl;
}
```

local variables of the function are destroyed

```
void fooB( ) {
        static int counter = 0;
        ++counter;
        cout << counter << endl;
}
```

static local variables are permanently allocated

# Static Local Variables

- After a function is finished, all the local variables of the function are destroyed.
- What do we do so that we can retain the value stored in local variables?
- Static local variables are permanently allocated for the entire lifetime of the program.

```
void fooA( ) {
        int counter = 0;
        ++counter;
        cout << counter << endl;
}
```

local variables of the function are destroyed

```
void fooB( ) {
        static int counter = 0;
        ++counter;
        cout << counter << endl;
}
```

static local variables are permanently allocated

# Pass by Value

Pass the value of the argument to the parameter of a function.

```
void k( int b ) {
        b = 5;
}
void j( ) {
        int c = 0;
        k( c );
}
```

# Pass by Value

Pass the value of the argument to the parameter of a function.

```
void k( int b ) {           // pass-by- A1
        b = 5;              // the value of the parameter is A2
}
void j( ) {
        int c = 0;
        k( c );             // c is A3
}
```

Affected or not?

# Pass by value

```
//Write a function to swap the values of two variables.
void swap( int a, int b ){
    int tmp = a;
    a = b;
    b = tmp;
}
```

# Pass by value

```
//Write a function to swap the values of two variables.
void swap( int a, int b ){
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
a = 5; b = 15;

tmp =5;

a = 15;

b = 5
```

# Pass by value

//Write a function to swap the values of two variables.

```
void swap( int a, int b ){

    int tmp = a;

    a = b;

    b = tmp;

}
```

```
a = 5; b = 15;

tmp =5;

a = 15;

b = 5
```

```
void foo ( ) {
   int x = 5, y =15;
   swap( x, y );

   cout << “x:” << x << endl;
   cout << “y:” << y << endl;
}
```

//fail to swap the values of the original variables associated with a and b.

# Pass by value

//Write a function to swap the values of two variables.

```
void swap( int a, int b ){

    int tmp = a;

    a = b;

    b = tmp;

}
```

```
a = 5; b = 15;

tmp =5;

a = 15;

b = 5
```

```
void foo ( ) {
   int x = 5, y =15;
   swap( x, y );

   cout << “x:” << x << endl;
   cout << “y:” << y << endl;
}
```

//fail to swap the values of the original variables associated with a and b.

x and y are not A1

# Pass by value

//Write a function to swap the values of two variables.

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

```
a = 5; b = 15;

tmp =5;

a = 15;

b = 5
```

The formal parameters and local variables are affected.

```
void foo ( ) {
   int x = 5, y =15;
   swap( x, y );

   cout << “x:” << x << endl;
   cout << “y:” << y << endl;
}
```

//fail to swap the values of the original variables associated with a and b.

x and y are not A1

# Reference Variables

int b = 10; int &a = b; // a is an A1 of b. a and b are the A2

# Reference Variables

int b = 10; int &a = b; // a is an alias of b. a and b are the same.

a is a A1 variable

# Reference Variables

Can be used as a function parameter to reference the original variable.

An alias for a variable.

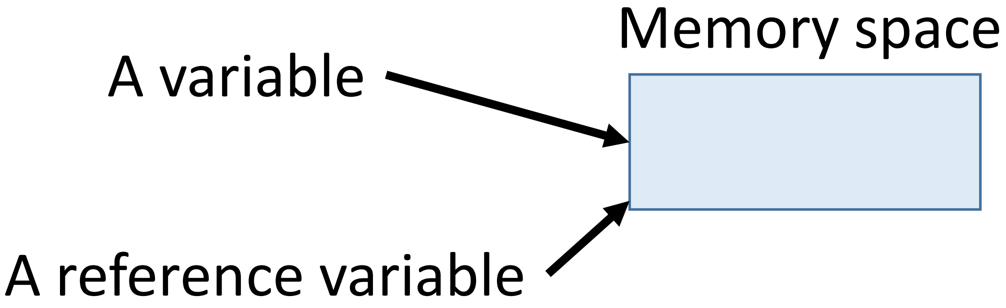
Any changes made through a reference variable are performed on the original variable.

```
int b = 10; int &a = b; // a is an alias of b. a and b are the same.
```

a is a reference variable

# Pass By Reference

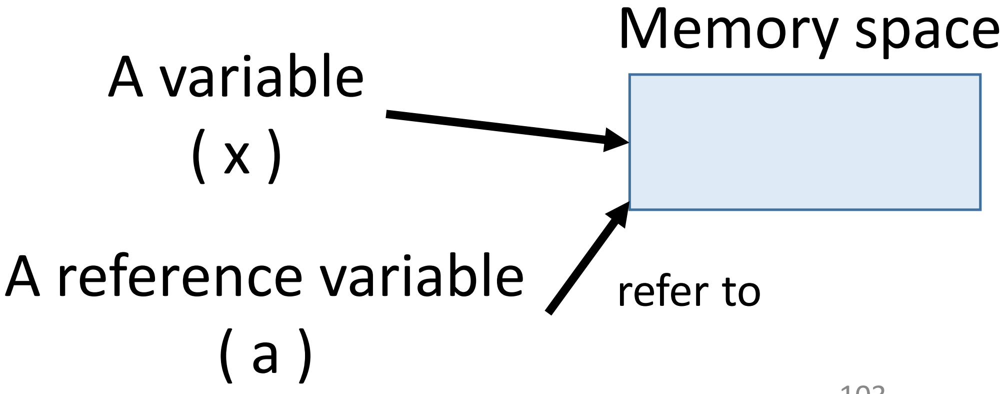
- A reference variable is used to refer to the same memory space of another variable.



# Pass By Reference

- A reference variable is used to refer to the same memory space of another variable.

```
void foo ( int & a, int & b) {
  a = 10;
  b = 20;
}
void main ( )  {
    int x = 6;
    int y = 7;

  foo( x, y );
    cout << x << “\t” << y << endl;
}
```



Memory space

A variable
( x )

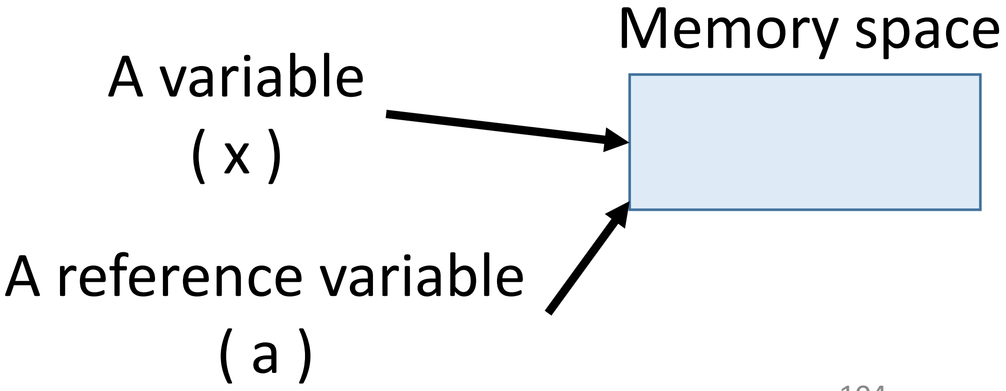
A reference variable
( a )

refer to

# Pass By Reference

- A reference variable is used to refer to the same memory space of another variable.
- We can use a reference variable as a parameter in a function.
- The parameter is an A1 for the actual parameter.
- If the value of a reference variable is changed, the value of the A2 variable is changed too.

```
void foo ( int & a, int & b) {
  a = 10;
  b = 20;
}
void main ( )  {
    int x = 6;
    int y = 7;

  foo( x, y );
    cout << x << "\t" << y << endl;
}
```



Memory space

A variable
( x )

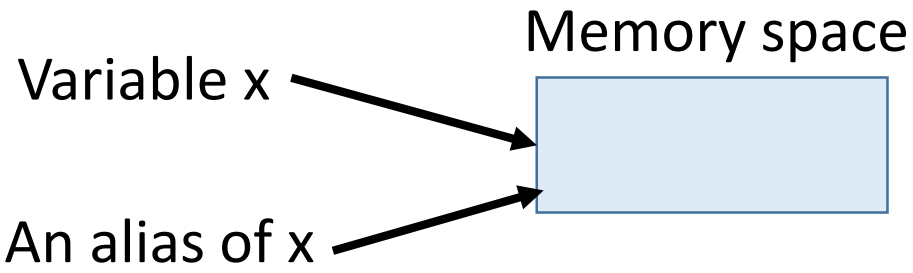
A reference variable
( a )

# Pass By Reference

- ➢ A reference variable is used to refer to the same memory space of another variable.
- ➢ We can use a reference variable as a parameter in a function.
- ➢ The parameter is an **alias** for the actual parameter.
- ➢ If the value of a reference variable is changed, the value of the original variable is changed too.

```
void foo ( int & a, int & b) {
  a = 10;
  b = 20;
}
void main ( )  {
    int x = 6;
    int y = 7;
    foo( x, y );
    cout << x << “\t” << y << endl;
}
```

```
void main ( )  {
    int x = 6;
    int &y = x; // alias
    y = 10;
    cout << x << “\t” << y << endl;
}
```



# Pass by Reference

```
// swap the values of the variables associated with a and b.
void swap(int &a, int &b){
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
a = 5; b = 15;

tmp =5;

a = 15;

b = 5
```

```
void foo ( ) {
   int x = 5, y =15;
   swap( x, y );

   cout << “x:” << x << endl;
   cout << “y:” << y << endl;
}
```

# Pass by Reference

// swap the values of the variables associated with a and b.

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

```
a = 5; b = 15;

tmp =5;

a = 15;

b = 5
```

```
void foo ( ) {
   int x = 5, y =15;
   swap( x, y );

   cout << "x:" << x << endl;
   cout << "y:" << y << endl;
}
```

x and y are A1 accordingly.

# Constant Reference Parameters

```
// Return the max between two numbers
int max(const int& num1, const int& num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

# Constant Reference Parameters

```
// Return the max between two numbers
int max(const int& num1, const int& num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Inside the function, the constant reference parameters cannot be changed.
No new objects are created for the constant reference parameters. The constant reference parameters are aliases of the actual parameters.

# Constant Reference Parameters

```
// Generalize the idea to any data type
```

```
int max(
  const int& num1
, const int& num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Inside the function, the constant reference parameters cannot be changed.
No new objects are created for the constant reference parameters. The constant reference parameters are aliases of the actual parameters.

# Constant Reference Parameters

```
// Return the max between two numbers
A max(
  const A& num1
  , const A& num2)
{
  A result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

```
int max(
  const int& num1
, const int& num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Inside the function, the constant reference parameters cannot be changed.
No new objects are created for the constant reference parameters. The constant reference parameters are aliases of the actual parameters.

# Constant Reference Parameters

```
// Return the max between two numbers
A max(
  const A& num1
  , const A& num2)
{
  A result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

What is the potential problem?

```
int max(
  const int& num1
, const int& num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Inside the function, the constant reference parameters cannot be changed.
No new objects are created for the constant reference parameters. The constant reference parameters are aliases of the actual parameters.

# Constant Reference Parameters

```
// Return the max between two numbers
A max(
  const A& num1
  , const A& num2)
{
  A result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

What is the potential problem?

The operators
A::> and
A::=
must be implemented.

```
int max(
  const int& num1
, const int& num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Inside the function, the constant reference parameters cannot be changed.
No new objects are created for the constant reference parameters. The constant reference parameters are aliases of the actual parameters.

# Constant Reference Parameters

```
// Return the max between two numbers
A max(
  const A& num1
  , const A& num2)
{
  A result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

What is the potential problem?

The operators
A::> and
A::=
must be implemented.

```
int max(
  const int& num1
, const int& num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Inside the function, the constant reference parameters cannot be changed.
No new objects are created for the constant reference parameters. The constant reference parameters are aliases of the actual parameters.

# Constant Reference Parameters

```
// Return the max between two numbers
A max(
   const A& num1
   , const A& num2)
{
   A result;
   if (num1 > num2)
     result = num1;
   else
     result = num2;
   return result;
}
```

What is the potential problem?

The operators
A::> and
A::=
must be implemented.

copy-constructor will be invoked here!

```
int max(
   const int& num1
,  const int& num2)
{
   int result;
   if (num1 > num2)
     result = num1;
   else
     result = num2;
   return result;
}
```

Inside the function, the constant reference parameters cannot be changed.
No new objects are created for the constant reference parameters. The constant reference parameters are aliases of the actual parameters.

# Constant Reference Parameters

```
// Return the max between two numbers
A max(
const A& num1
, const A& num2)
{
  A result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Return a “value”.

What is the potential problem?

The operators
A::> and
A::=
must be implemented.

copy-constructor will be invoked here!

```
int max(
  const int& num1
, const int& num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Inside the function, the constant reference parameters cannot be changed.
No new objects are created for the constant reference parameters. The constant reference parameters are aliases of the actual parameters.

# Function Abstraction

A function body can be treated as a black box that contains the function implementation.

```
int min( int num1, int num2 )
{
        int min_value;
        min_value = num1;
        if ( num2 < num1 )
                { min_value = num2; }
        return min_value;
}
```

# Function Abstraction

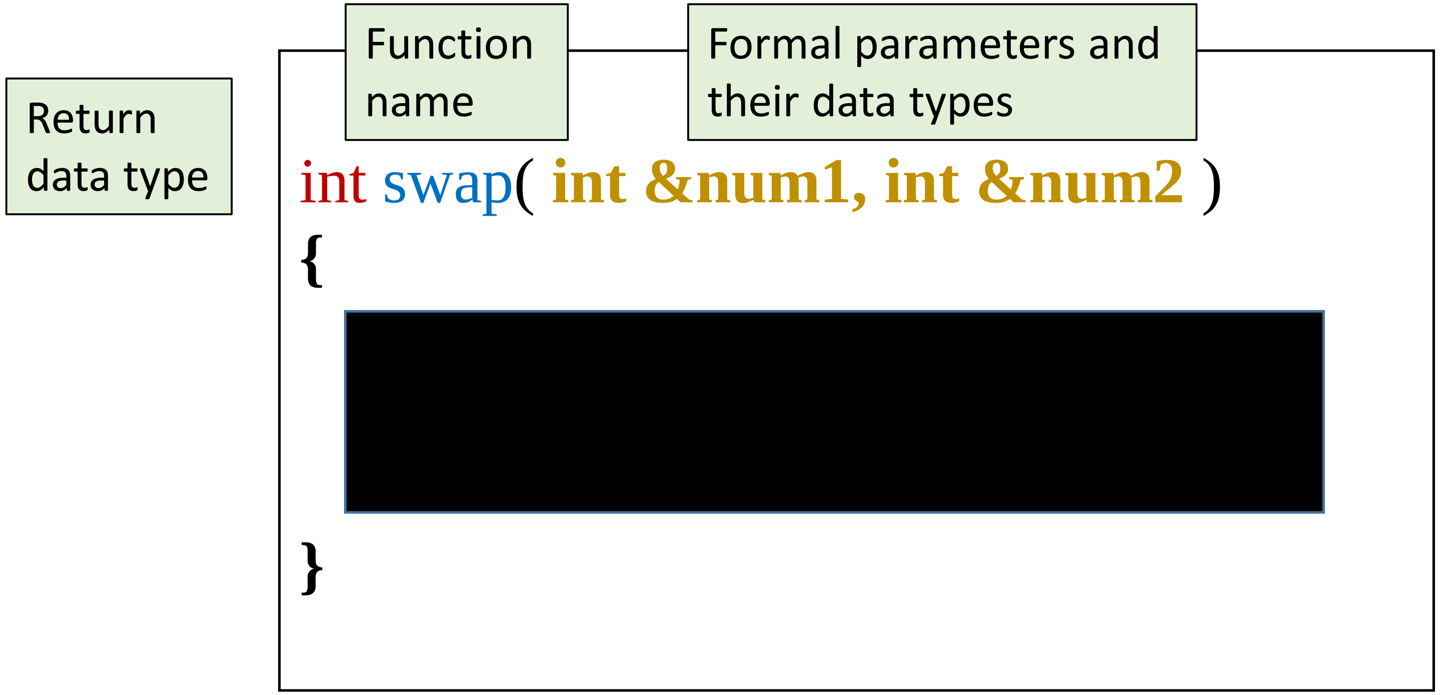
A function body can be treated as a black box that contains the function implementation.

```
int swap( int &num1, int &num2 )
{
    [redacted]
}
```

# Function Abstraction

A function body can be treated as a black box that contains the function implementation.

We only need to know



Return data type

Function name

Formal parameters and their data types

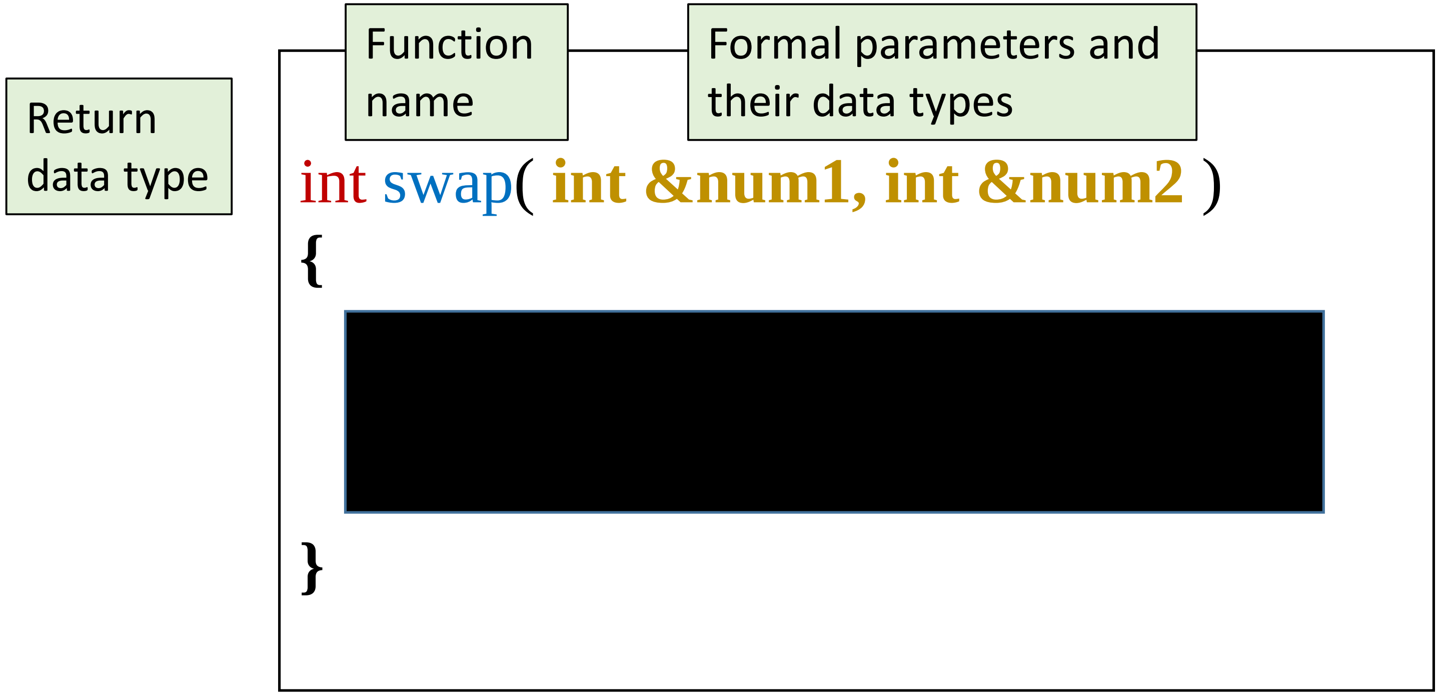
```
int swap( int &num1, int &num2 )
{
[redacted]
}
```

# Function Abstraction

A function body can be treated as a black box that contains the function implementation.

We only need to know

- A1 data type
- The function A2
- A3 parameters and their A4



Return data type

Function name

Formal parameters and their data types

```
int swap( int &num1, int &num2 )
{

}
```

# Benefits of Functions

- Write a function once and reuse it anywhere
- Hide the information of functions
- Hide the implementation from clients.
- Reduce program complexity

```
int swap( int &num1, int &num2 )
{
  [redacted]
}
```

```
void test( ) { // test is a "client" of swap
   int x = 4, y = 7;
   int z = swap( x, y);
}
```

# Benefits of Functions

- Write a function once and A1 it anywhere
- A2 the information of functions
- A3 the implementation from clients.
- Reduce program complexity

```
int swap( int &num1, int &num2 )
{
    [redacted]
}
```

```
void test( ) { // test is a “client” of swap
    int x = 4, y = 7;
    int z = swap( x, y);
}
```

# Example

System Specification

- ➢Implementation a program to ask to input the number of students.
- ➢Input their scores and then output the average score.
- ➢Finally, sort their scores in ascending order and display the result.

# Example

System Specification

- Implementation a program to ask to input the number of students.
- Input their scores and then output the average score.
- Finally, sort their scores in ascending order and display the result.

```
void processStudentScore( ) {




}
```

# Example

System Specification

- Implementation a program to ask to input the number of students.
- Input their scores and then output the average score.
- Finally, sort their scores in ascending order and display the result.

```
void processStudentScore( ) {
   askForInput( );
   computeAverageScore( );
   sortScore( );


}
```

# Example

System Specification

- Implementation a program to ask to input the number of students.
- Input their scores and then output the average score.
- Finally, sort their scores in ascending order and display the result.

```
void processStudentScore( ) {
   askForInput( );  // number of students? Input scores
   computeAverageScore( );        // output?
   sortScore( );                  // output?

}
```

# Function abstraction
# Stepwise refinement

System Specification

- ➢Implementation a program to ask to input the number of students.
- ➢Input their scores and then output the average score.
- ➢Finally, sort their scores in ascending order and display the result.

```
void processStudentScore( ) {
   askForInput( );  // number of students? Input scores
   computeAverageScore( );        // output?
   sortScore( );                  // output?


}
```

```
void processStudentScore( ) {
            askForNumberofStudents( );
            askForScoreInput( );
            computeAverageScore( );
            outputAverageScore( );
            sortScore( );
            outputScore( );
}
```

# Function abstraction
# Stepwise refinement

System Specification

- ➢Implementation a program to ask to input the number of students.
- ➢Input their scores and then output the average score.
- ➢Finally, sort their scores in ascending order and display the result.

```
void processStudentScore( ) {
            askForNumberofStudents( );
            askForScoreInput( );
            computeAverageScore( );
            outputAverageScore( );
            sortScore( );
            outputScore( );
}
```

```
void computeAverageScore( ) {
  mAverage = 0;
  if (mNumberOfStudents <=0 ) return;
  double total = computeTotalOfAllScores( );
  mAverage = total / mNumberOfStudents;
}
```

# Function abstraction
# Stepwise refinement

**Benefits**

- Simpler Program
- Reusing Functions
- Easier Developing, Debugging, and Testing
- Better Facilitating Teamwork

```
void processStudentScore( ) {
            askForNumberofStudents( );
            askForScoreInput( );
            computeAverageScore( );
            outputAverageScore( );
            sortScore( );
            outputScore( );
}
```

```
void computeAverageScore( ) {
  mAverage = 0;
  if (mNumberOfStudents <=0 ) return;
  double total = computeTotalOfAllScores( );
  mAverage = total / mNumberOfStudents;
}
```

# Function abstraction
# Stepwise refinement

➢ We should adopt it to develop programs.

➢ When writing a program:

- use the divide and conquer strategy
- adopt stepwise refinement to decompose a program/function into subproblems.

➢ Decompose subproblems smaller when necessary

→ Increase ability to manage

More flexible

```
void processStudentScore( ) {
            askForNumberofStudents( );
            askForScoreInput( );
            computeAverageScore( );
        ➡   outputAverageScore( );
        ➡   sortScore( );
            outputScore( );
}
```

```
void computeAverageScore( ) {
   mAverage = 0;
   if (mNumberOfStudents <=0 ) return;
   double total = computeTotalOfAllScores( );
   mAverage = total / mNumberOfStudents;
}
```

# Function abstraction
# Stepwise refinement

- We should adopt it to develop programs.

- When writing a program:

  - use the divide and conquer strategy

  - adopt stepwise refinement to decompose a program/function into subproblems.

- Decompose subproblems smaller when necessary

  → Increase ability to manage

  More flexible

```
void processStudentScore( ) {
            askForNumberofStudents( );
            askForScoreInput( );
            computeAverageScore( );
      ➡     sortScore( );
      ➡     outputAverageScore( );
            outputScore( );
}
```

```
void computeAverageScore( ) {
   mAverage = 0;
   if (mNumberOfStudents <=0 ) return;
   double total = computeTotalOfAllScores( );
   mAverage = total / mNumberOfStudents;
}
```

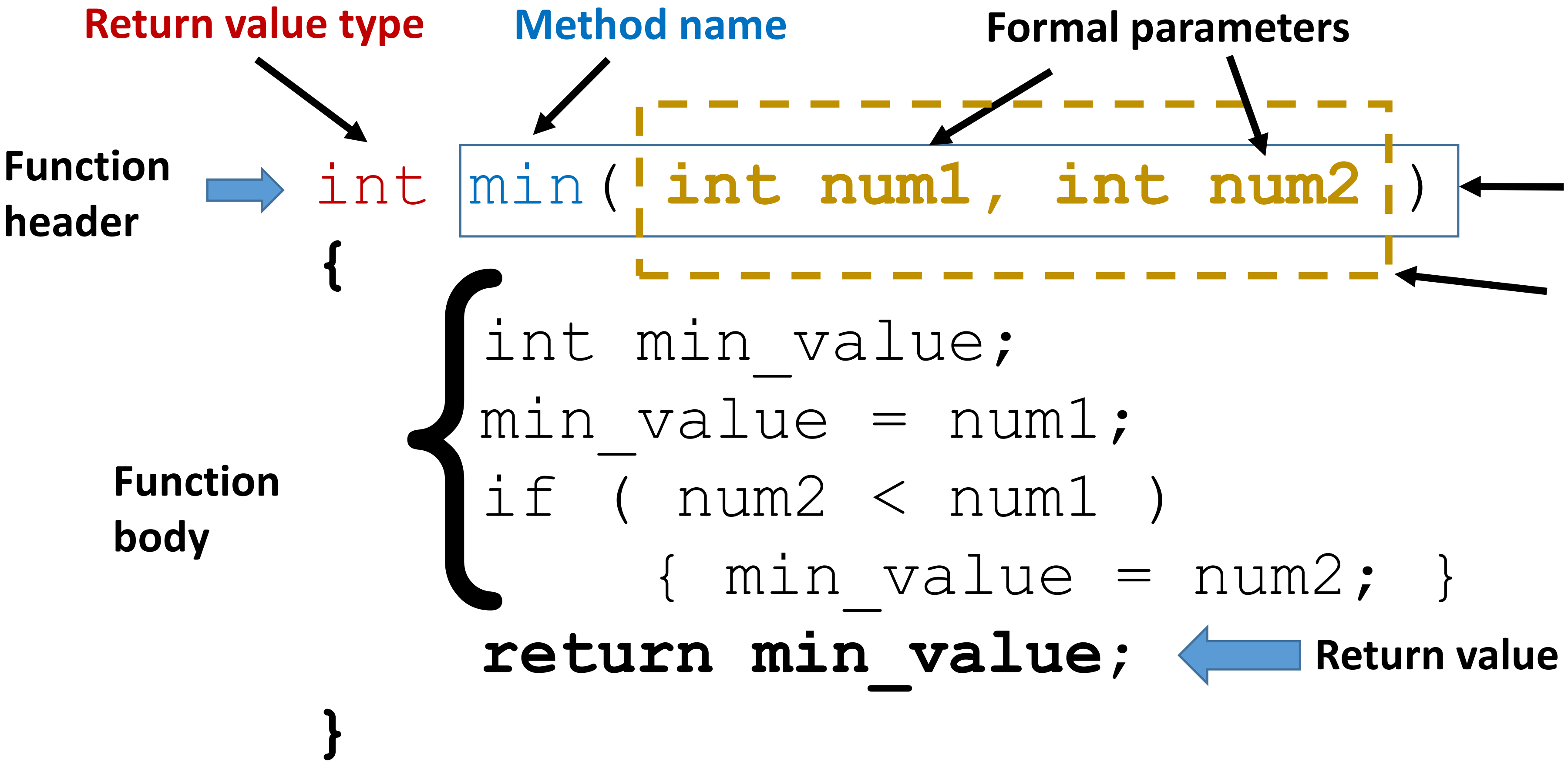
# Intended Learning Outcomes

- Describe functions and scopes
- Describe the process of function call
- Distinguish between formal parameters and actual parameters
- Distinguish between pass-by-value and pass-by-reference
- List the benefits of function abstraction

# Supplemental Material

# Function definition

A function is a collection of statements that are grouped together to perform an operation.



Return value type

Method name

Formal parameters

Function header

Function signature

Parameter list

Function body

Return value

```
int min( int num1, int num2 )
{
    int min_value;
    min_value = num1;
    if ( num2 < num1 )
         { min_value = num2; }
    return min_value;
}
```

int a = 15;
int x = min (-12, a);

actual parameters
(arguments)

**Function Body: the part enclosed by the outer most braces { and }.**

# Function definition

- Function signature is the combination of the function name and the parameter list.
- A formal parameter: A variable is defined in the function header.
- An actual parameter or an argument: The value that is passed to a formal parameter when a function is invoked.

# When we call a function, what happens?

```
int x = 0, y = 1;
void foo( int a, int &b ) // b is pass-by-reference
{
  a = 5;
  b = 2;
}

foo(y, x);
```

# Example
# System Requirement

- Implement a toy system.
- Show a menu with the following options:
  1. Input two numbers x and y
  2. Generate randomly the two numbers
  3. Show them
  4. Show their sum
  5. Swap the two numbers
  6. Show the number of times that each member function is called.
  7. Quit the program
- Implement all the options

# Example

Compute sum = (1*1+1*2+..1*100)
+
(2*1+2*2+...+2*100)
+ .......
(100*1+100*2+...+100*100)

# Example

Compute sum = (1*1+1*2+..1*100)

+

(2*1+2*2+...+2*100)

+ …….

(100*1+100*2+...+100*100)

```
int j = 0;
int sum = 0;
for (int i = 0; i < 100; ++i ) {
          for (int i = 0; i < 100; ++j ) {
                    sum += i*j;
          }
}
```

# Compute sum = (1*1+1*2+..1*100)

+
(2*1+2*2+...+2*100)
+ .......
(100*1+100*2+...+100*100)

```
int sum = 0;
const int num = 100;
for (int i = 0; i < num ; ++i ) {
        for (int j = 0; j < num ; ++j ) {
                sum += i*j;
        }
}
```

# Remarks

Read the instruction carefully.
While you implement your programs, you gradually forget about the instruction.
Always check it after a while.

# Remarks

Write a program to read a and b. Display:

a + b

a – b

After a while: you might print a –b and then a + b.

# Exercise
# Parameter order association

```
void f(int &a, int &b, int &c); // forward declaration

void foo( ) {
	int a = 3;
	int b = 2;
	int c = 1;
	f(c, b, a);
	…
}
```

what’re the mappings between the actual parameters and formal parameters?

# Exercise
# Parameter order association

```
void f(int &a, int &b, int &c, int &d) {
      a = 5; b = 6; c = 7; d = 8;
}

void foo( ) {
      int a = 3;
      int b = 2;
      int c = 1;
      int d = 0;
      f(d, c, b, a);
      cout << a << \t” <<b << “\t” << c << “\t” << d << endl;
}
```

What is the output after foo is finished.

# Inline Functions

Function calls involve runtime overhead:
1) pushing arguments and CPU registers into the stack
2) transferring control to and from a function

- C++ provides inline functions to **avoid function calls**.
- The compiler copies the function code in line at the point of each invocation.
- Inline functions are not called.

May not be faster than non-inline functions.

```
inline void foo( )  { //body }
```

# Inline Functions

C++ provides inline functions to **avoid function calls**.

- ➢ The compiler copies the function code in line at the point of each invocation.
- ➢ Inline functions are not called.

```
inline void foo( )  {
        cout << “here” << endl;
}
void g( ) {
        foo( );
        foo( );
}
```

```
void g( ) {
        cout << “here” << endl;
        cout << “here” << endl;
}
```

# Scope of Local Variables

Any problem?

```
void foo( )
{
    int b = 20;
    for ( int i = 0; i < 90; ++i) {
        ……
    }
    for ( int i = 0; i < 10; ++i) {
        int a = 10;
        ……
    }
}
```

# Scope of Local Variables

**Any problem?**

```
void foo( )
{
   int b = 20;
   for ( int i = 0; i < 90; ++i) {
       for (int j = 0; j < 10; ++i) {
           int a = 10;
           ……
      }
   }
}
```

# Scope of Local Variables

## Any problem?

Counter update problem

```
void foo( )
{
    int b = 20;
    for ( int i = 0; i < 90; ++i) {

        for (int j = 0; j < 10; ++i) {
            int a = 10;
            ......
        }
    }
}
```

# Constant Reference Parameters

If we use pass-by-value, the original variables are not changed in a function too.

**Why do we want to use constant reference parameters?**

```
// Return the max between two numbers
```

```
int max( A num1, A num2)
{
        ……
  return result;
}
```

```
int max( const A & num1, const A & num2)
{
        ……
  return result;
}
```

# Short Functions
# Compiler Decision

- Inline functions are desirable for short functions.
- Not suitable for long functions that are called in multiple.
- This is because long inline functions will dramatically increase the executable code size.
- The same code appears in multiple places.
- The compilers ignores the inline keyword if the function is too long.

# Modular code

Code is separated into independent modules.

Internal details of individual modules are hidden.

Example: Monte Carlo Simulation

```
vector3 computeOneSamplePoint( ……)
vector<vector3> computeSamplePoints(int numSamples)
void displaySamplePoints(const vector<vector3> &samplePoints)
```