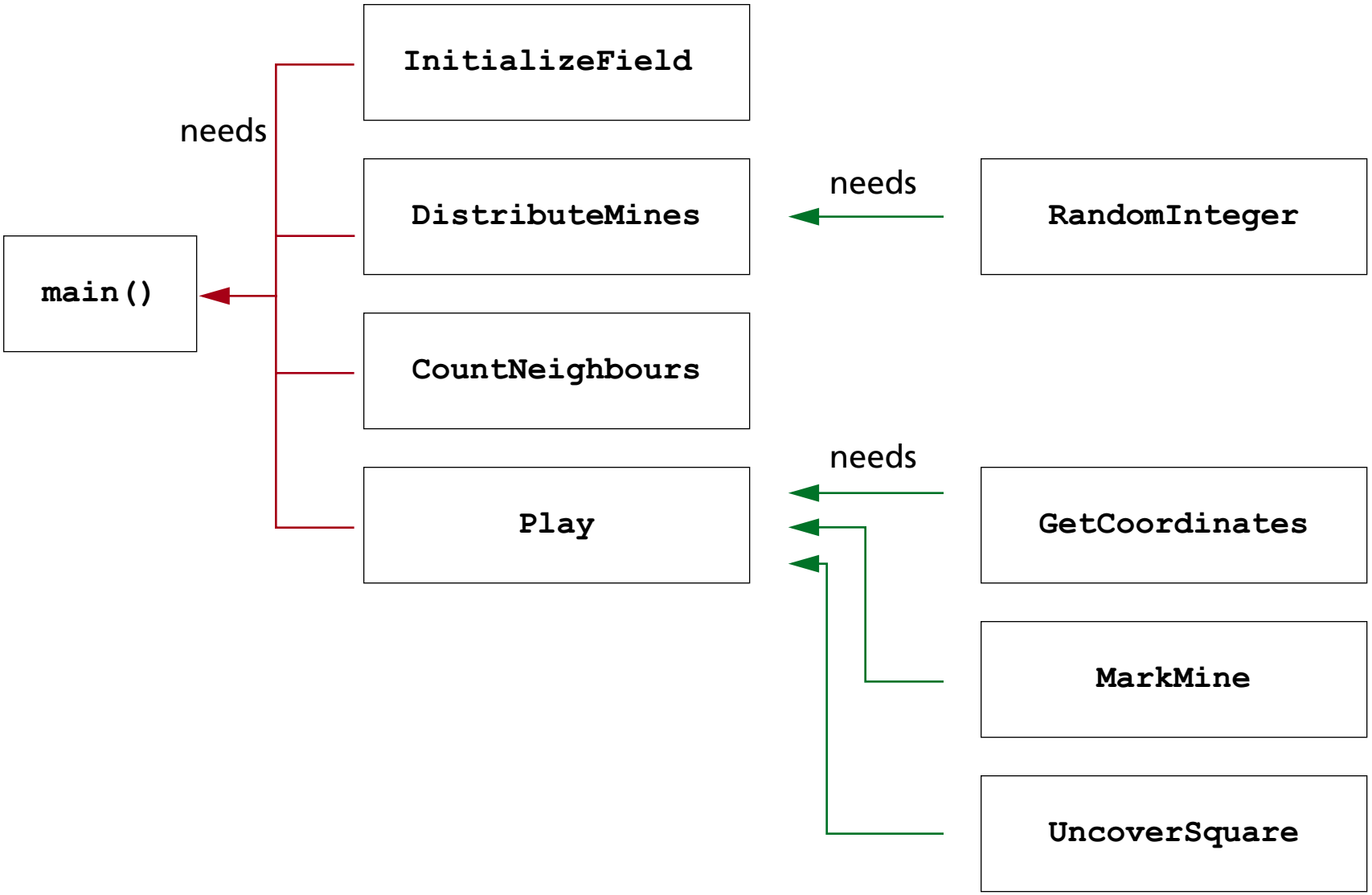
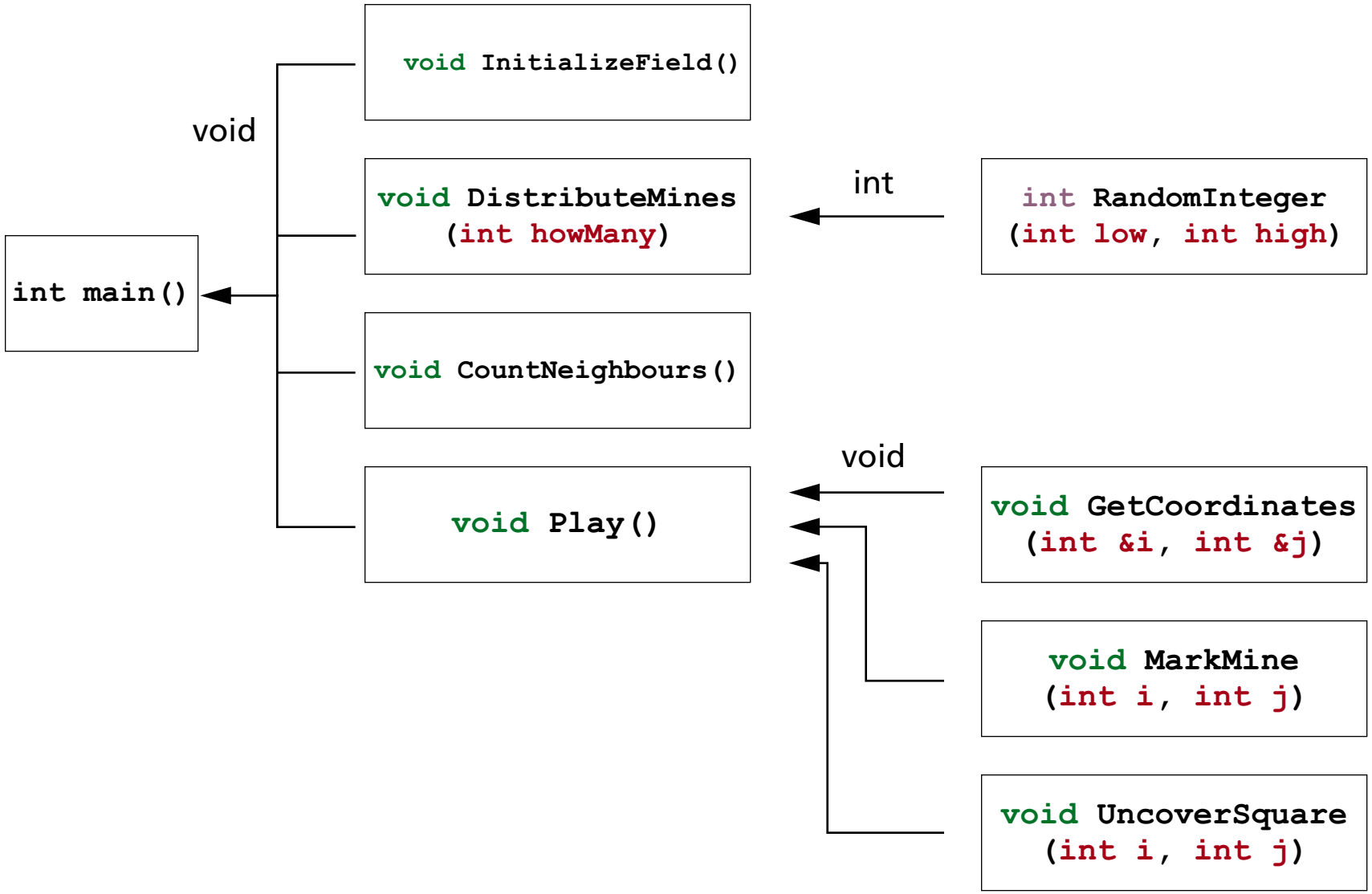
Chapter 11

Using Functions for Procedural Abstraction (Part 1)

The functional structure of Mine Sweeper:



The functional (C++) structure of Mine Sweeper:



11.1 Function Prototypes

In C++, all functions must be declared before they are used.

Typically, this is accomplished by use of a **function prototype**.

Prototypes specify the data interface of a function by …

- its return type,
- the type of its parameters, and
- the number of its parameters.

Example:

```
int square(int i);
```

The use of parameter names in a prototype is optional:

```
int square(int);
```

General Form of a Function Prototype Definition

Prototype definitions have to appear before a function is used in a program.

A function prototype definition is the same as a function definition, except that no body is present:

type ***function_name***(*type* ***parameter_name***$_1$,

type ***parameter_name***$_2$,

...

type ***parameter_name***$_N$);

Prototypes provide information for the compiler

Prototypes allow the compiler to perform three important operations, namely:

- Detect differences between the **number of arguments** used to call a function and the number of parameters in the function.
- Find and report any **illegal type conversions** between the type of arguments used to call a function and the type of definition of its parameters.
- What type of **code to generate** when a function is called. Different return types must be handled differently by the compiler.

The following program uses a function prototype to enforce type checking:

```
float square(int i); // prototype

void main()
{
  int i, x = 10; float s[x];

  for (i=0; i < x; i++)
    s[i] = square(i);
}

float square(int i)
{
  return(float(i * i));
}
```

11.2 Scope Rules of Functions

The **scope rules** of a programming language are the rules that govern how an object—a variable, a data structure, a function—may be accessed by various parts of a program.

The scope rules determine …

- what code has access to a variable and
- the lifetime of a variable.

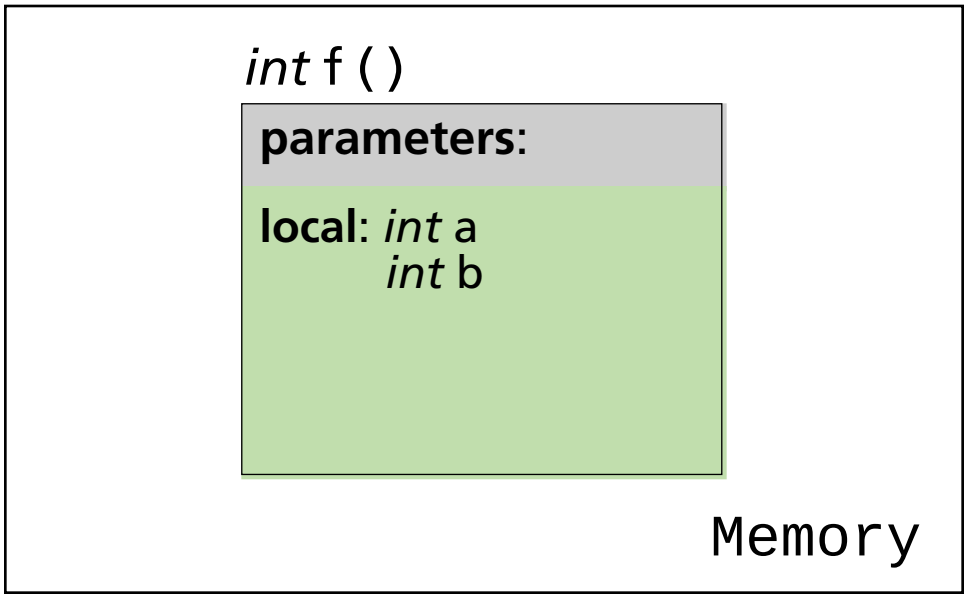
There are three **types of variables**:

- **local** variables,
- **global** variables, and
- **formal** parameters.

11.2.1 Local Variables

Variables that are declared inside a function are called **local variables**.

```
int f()
{
   int a, b;
   ...
}
```



In this example, **a** and **b** are local variables of the function **f()**. That is, the function body is the the **scope** of variables **a** and **b**.

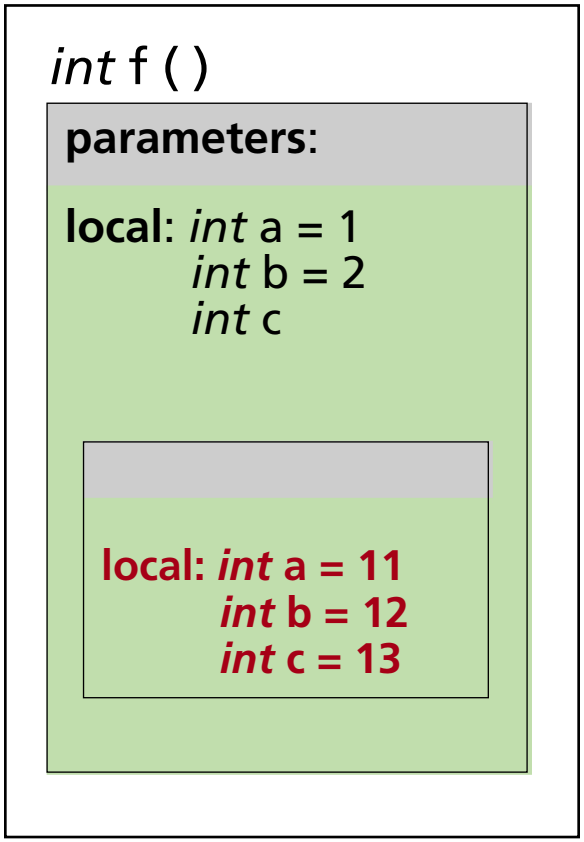
Variables can also be localized to a **block**[1] (example 1):

```
int f()
{
  int a=1, b=2, c;

  cout << c = a * b;
  {
    int a=11, b=12, c=13;

    cout << a * b << ":";
    cout << c;
  }

  cout << a * b << ":";
  cout << c; }
```



1. A block begins with an opening curly brace and ends with a closing curly brace.

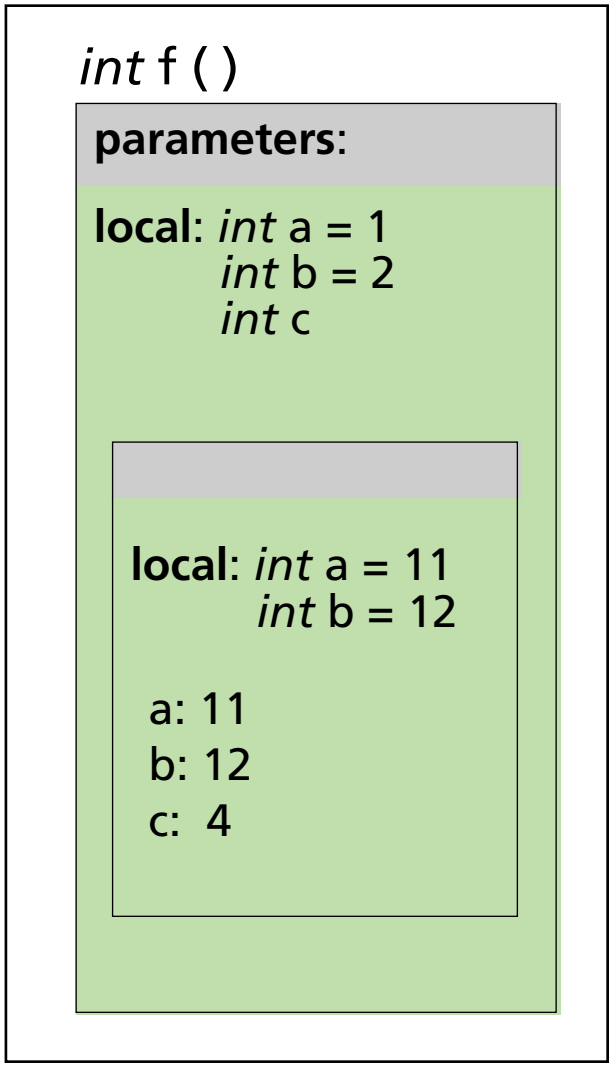
Scopes of variables (example 2):

```
int f()
{
  int a=1, b=2, c=4;

  cout << c = a * b;
  {
    int a=11, b=12;

    cout << a * b << ":";
    cout << c;
  }

  cout << a * b << ":";
  cout << c;
}
```



Examples of Blocks:

A block is any code section delimeted by opening and closing curly braces:

- { … }
- if (*condition*) { … }
- switch (*condition*) { … }
- while (*condition*) { … }
- do { … } while (*condition*)
- for (*init*; *condition*; *inc_dec*) { … }
- *type function* (*arguments*) { … }
- …

Local variables …

- are **created** upon entry into a block
- are local to this block
- are **destroyed** upon exit from this block

Hence, local variables exist only while the block of code in which they are declared is executing.

… and functions

- Each function defines its own scope.
- All variables needed within a function should be declared at the beginning of that function´s code block.
- The variables declared within one function have no effect on those declared in another—even if the variables share the same name.

11.2.2 Formal Parameters and Local Variables

If a function uses arguments, it must declare variables that will accept the values of those arguments.

These variables are called the **formal parameters** of the function.

- A formal parameter behaves like any other local variable inside the function.
- The scope of a formal parameter is local to its function.

Example:

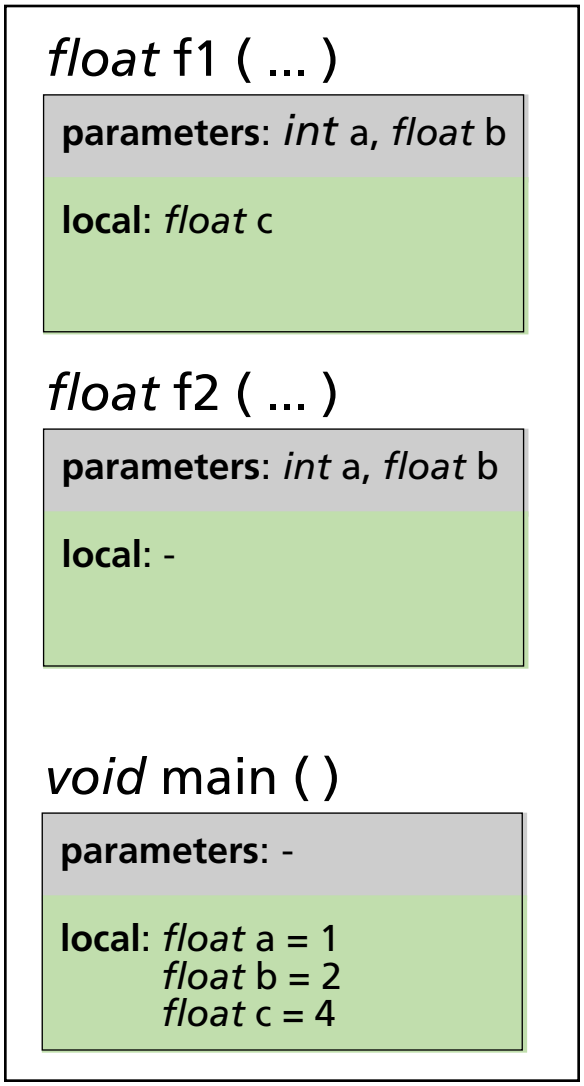
```
int f(int a, float b, double c) { ... }
```

This function has three formal parameters **`a`**, **`b`**, and **`c`** of type **integer**, **float**, and **double**.

```
float f1(int a, float b)
{
  float c = a + b;
  return c;
}

float f2(int a, float b)
{
  c = a + b;
  return 2*c;
}

void main()
{
  float a=1, b=2, c=5;
  cout <<
    f1(a,b) + f2(a,c) + c; }
```

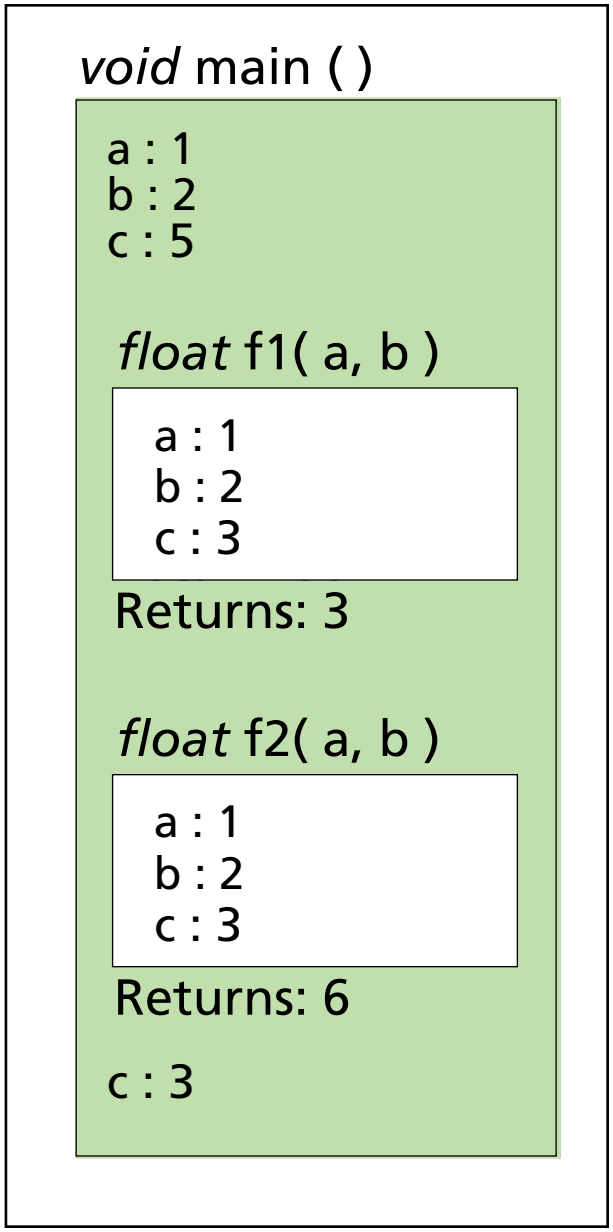


During execution of the main() function we get the following variable scopes and values:

```
float f1(int a, float b)
{
   float c = a + b;
   return c;
}

float f2(int a, float b)
{
   c = a + b;
   return 2*c;
}

void main()
{
   float a=1, b=2, c=5;
   cout <<
      f1(a,b) + f2(a,c) + c; }
```



11.3 Global Variables

Global variables …

- are known throughout the entire program,
- can be used by any piece of code,
- maintain their values during the entire execution of the program.

The scope of global variables extends to the entire program.

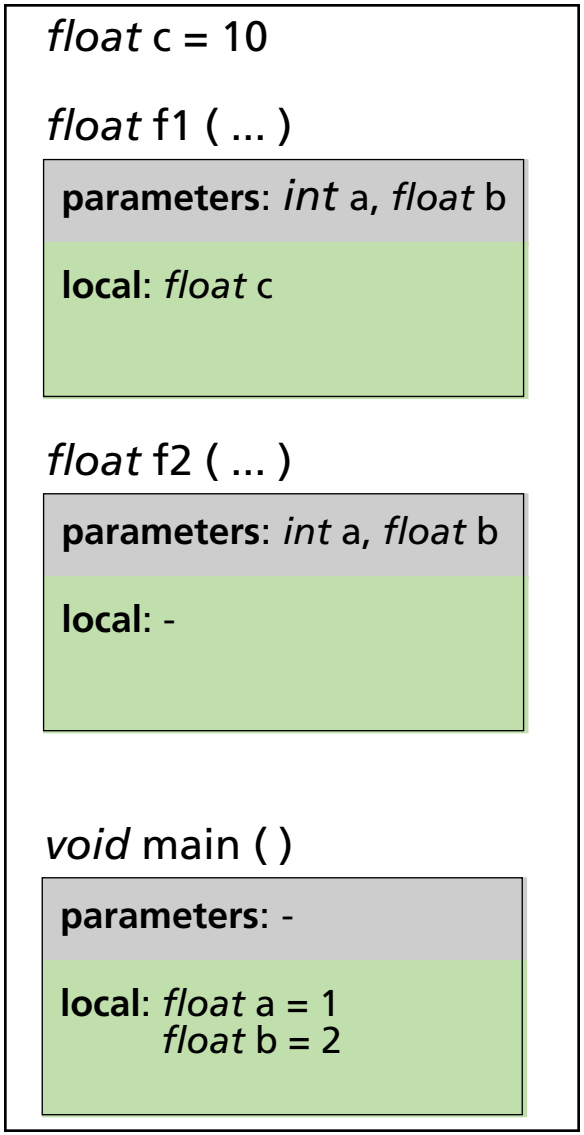
Global variables are created by declaring them outside of any function.

When a global and a local variable share the same name, the **local** variable has precedence.

```
float c = 10;
float f1(int a, float b)
{
  float c = a + b;
  return c;
}

float f2(int a, float b)
{
  c = a + b; return 2*c;
}

void main()
{
  float a=1, b=2;
  cout <<
  f1(a,b) + f2(a,c) + c;}
```

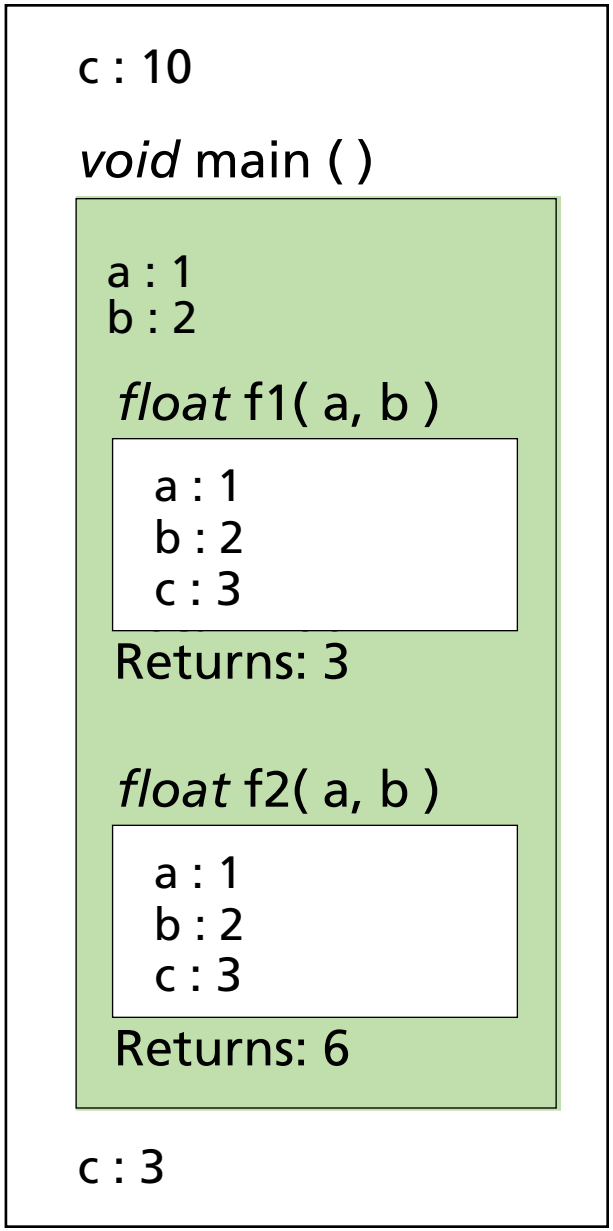


During execution of the main() function we get the following variable scopes and values:

```
float c = 10;
float f1(int a, float b)
{
   float c = a + b;
   return c;
}

float f2(int a, float b)
{
   c = a + b; return 2*c;
}

void main()
{
   float a=1, b=2;
   cout <<
      f1(a,b) + f2(a,c) + c; }
```



11.4 The *return* Statement

11.4.1 Returning from a Function

A function returns to its calling routine either …

- when the function´s closing curly brace is encountered
- or when a **return** statement is executed.

The **return** statement can be used with or without an associated value.

However, functions that are declared as returning a value (with non-**void** return type) must return a value.

Only functions declared as **void** can use **return** without any value.

11.4.2 Functions Returning *void*

For **void** functions, the **return** statement is mostly used as a program-control structure, e.g., to prevent part of a function from executing, as the following example demonstrates.

```
void power(int base, int exp)
{
  int i = 1;

  if(exp < 0) return; // No negative exponents
                      // Bypass rest of function
  for( ; exp; exp--) i = base * i;

  cout << base << "^" << exp << " is " << i;
}
```

11.4.3 Returning Values

Every function of a non-**void** type has to return a value by the **return** statement.

Any of these value-returning functions can be used as an operand in an expression.

Examples:

```
z = power(x, y);

if(max(x, y) > 100) cout << "...";

switch(abs(x)) { ... }
```

The function **power** as a value-returning function:

```
int power(int base, int exp)
{
  int i = 1;

  if(exp < 0) return(0); // No negative exponents

  for( ; exp; exp--) i = base * i;

  return(i);
}

void main()
{
  int z;
  z = power(2,3); ... }
```

11.5 References

- G. Blank and R. Barnes, *The Universal Machine*, Boston, MA: WCB/ McGraw-Hill, 1998. Chapter 9.
- H. Schildt, C++ from the Ground Up, McGraw-Hill, Berkeley, CA, 1998. Chapter 7.