Hochiminh City University of Technology
Computer Science and Engineering
[CO1027] - Fundamentals of C++ Programming

Function

Lecturer: Duc Dung Nguyen
Credits: 3

Outcomes

- Solving problems with functions

Outline

- Function: definition, declaration, parameters, returned value
- Scope of variables
- Storage

Function

Function

- You should never write monolithic code
  - Difficult to write correctly.
  - Difficult to debug.
  - Difficult to extend.
- Hard to maintenance
- Non-reusable
- Nonsense!

Function

- Definition: a group of statements that is given a name, and which can be called from some point of the program.

- All C++ functions (except the special case of the main function) should have:
  - A declaration: this is a statement of how the function is to be called.
  - A definition: this is the statement(s) of the task the function performs when called

Function Declaration

- A function is declared with the syntax:

```
returnVariableType  functionName(parameter1, parameter2,
...,parameterN);
```

- **Note the semi-colon at the end of the statement.**

Function Definition

- A function is defined with the syntax:

```
retVariableType  functionName(parameter1, parameter2,
...,parameterN)
{
     statement(s);
}
```

Function

- C++ functions can:
  - Accept parameters, but they are not required
  - Return values, but a return value is not required
  - Can modify parameters, if given explicit direction to do so

Function: No Return, No Parameters

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

void displayMessage();

int main() {
   displayMessage();
   return 0;
}

void displayMessage() {
   cout << "Welcome to CO1011!\n";
}
```

Functions with Parameters

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

void displaySum(int, int);

int main() {
   displaySum(5, 10);
   return 0;
}

void displaySum(int num1, int num2) {
   printf("%d + %d = %d\n", num1, num2, num1 + num2);
}
```

Functions with Return

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

int computeSum(int, int);

int main() {
   int sum = computeSum(5, 10);
   printf("%d + %d = %d\n", 5, 10, sum);
   return 0;
}

int computeSum(int num1, int num2) {
   return num1 + num2;
}
```

Inline Function

- Similar to function, except that the compiled code will be inserted where we call inline functions.
- Purpose: improve performance
- ```
  inline <return type> <function name>(<parameters>) {
      <function body>
  }
  ```
  ```

Pass Parameters

- Parameters: there are two ways to pass parameters to a function
  - Value: the value will be copied to local variable (parameter) of the function
  - Reference (only in C++): the parameter is associated with passed variable
    - Passing by reference refers to passing the address of the variable
    - Any change in the parameter affects the variable

Example

```
#include<iostream>
using namespace std;
void increment(int &input);

int main() {
   int a = 34;
   cout << "Before the function call a = " << a << "\n";
   increment(a);
   cout << "After the function call a = " << a << "\n";
   return 0;
}

void increment(int &input){
   input++;
}
```

Arrays as Parameters

- There are three methods for passing an array by reference to a function:
  - `returnType functionName(variableType *arrayName)`
  - `returnType functionName(variableType arrayName[arraySize])`
  - `returnType functionName(variableType arrayName[])`

Example

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

void arrayAsPointer(int *array, int size);

int main() {
    const int size = 3;
    int array[size] = { 33,66,99 };
    arrayAsPointer(array, size);
    return 0;
}

void arrayAsPointer(int *array, int size) {
    cout << setw(5);
    for (int i = 0; i<size; i++)
    cout << array[i] << " ";
    cout << "\n";
}
```

Default Parameters

- Default arguments are used in place of the missing trailing arguments in a function call.
- Default arguments must be the **rightmost** (trailing) arguments in a function's parameter list

Example

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

int computeBoxVolume(int length = 1, int width = 1, int height = 1);

int main() {
   cout << "The default box volume : " << computeBoxVolume() <<endl;
   cout << "The second box volume : " << computeBoxVolume(10) << endl;
}

int computeBoxVolume(int length, int width, int height) {
   return length * width * height;
}
```

Function Overloading

- Several functions can have the same name: overloaded functions.
- Functions with the same name have different signatures (prototypes).
- Function signature: name + parameter list
- The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call

Example

```
float add(float a, float b);
int add(int a, int b);
double add(int a, double b);

float add(float a, float b) {
   return a + b;
}

int add(int a, int b) {
   return a + b;
}

double add(int a, double b) {
   return (double)a + b;
}
```

Function in Header Files

- It is quite common to put functions into a header file.
  - It makes your main program look cleaner.
  - It makes your code reusable.
- In this case, function prototypes is **REQUIRED**.

Scope of Variables

Scope of Variables

- The portion of the program where a variable can be used is known as its scope.
- Global vs. Local variables
  - Global variables can be accessed everywhere in the program
  - Local variables can only be accessed inside the block where it is declared (local scope).
  - Local scope begins at the identifier's declaration and ends at the terminating right brace (})

Scope of Variables

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

int x = 5; // global variable

int main() {
    cout << "global x = " << x << endl;
    int y = 7; // local variable;
    cout << "local y in main's scope = " << y << endl;

    { // start a new scope
        int z = 9;
        cout << "local z in inner scope = " << z << endl;
    }
    return 0;
}
```

Unary Scope Resolution Operator

- It's possible to declare local and global variables of the same name.
- Local variables take precedence over global variables
- The scope resolution operator `::` is used to access to the global variable.

Example

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

int num = 8; // global variable

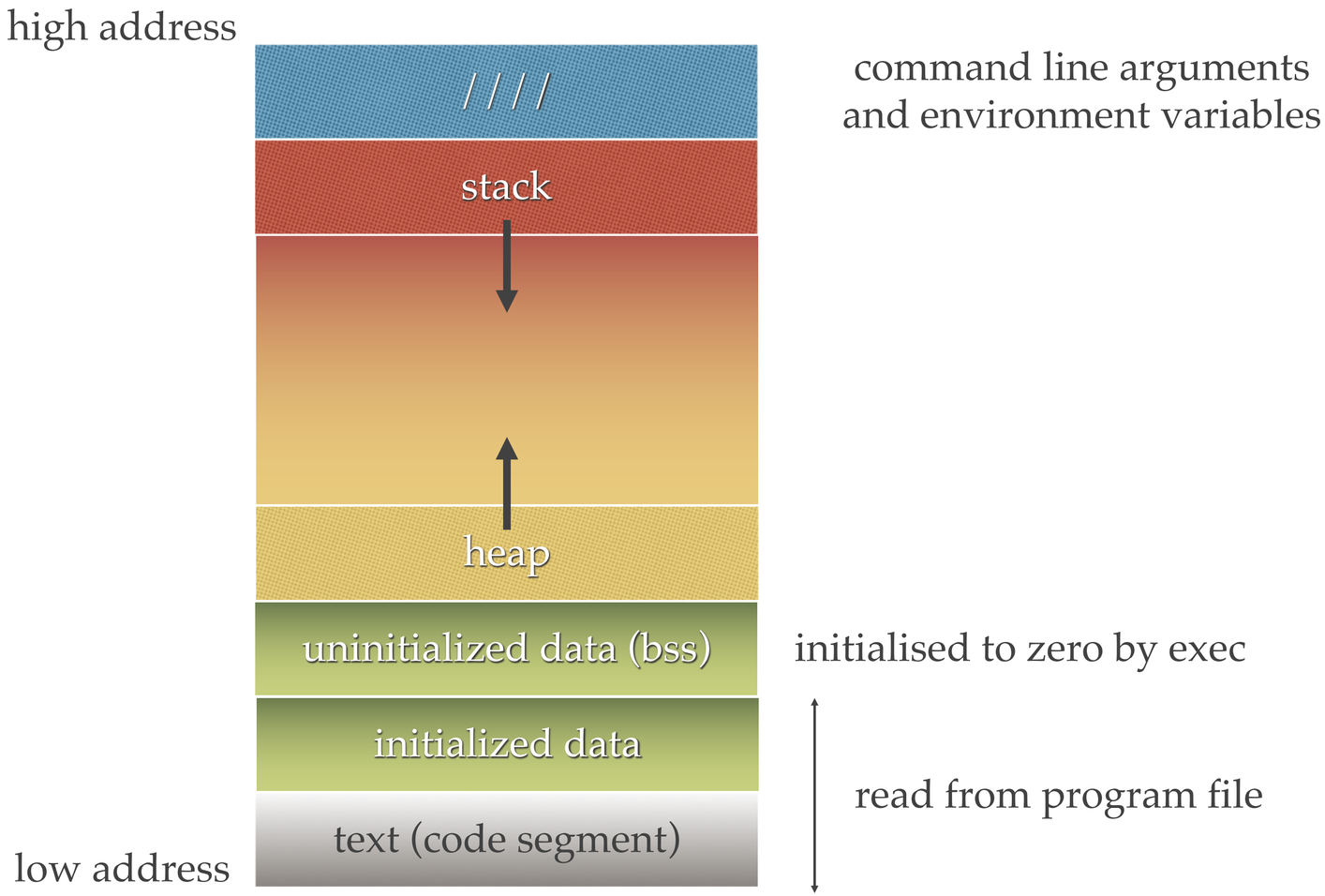
int main() {
   int num = 10; //local variable
   cout << "Local variable num = " << num << endl;
   cout << "Global variable num = " << ::num << endl;
   return 0;
}
```

Storage

Storage

- How your program is organized?
- What are common errors?

Storage


high address
////
stack
heap
uninitialized data (bss)
initialized data
text (code segment)
low address
command line arguments
and environment variables
initialised to zero by exec
read from program file

Storage

- Code segment: contains executable code (binary code)
- Data segment:
  - Initialized data: global, static, constants
  - Uninitialized data
- Heap: contains allocated memory at runtime
- Stack: stores local variables, passed arguments, and return address

Function Call

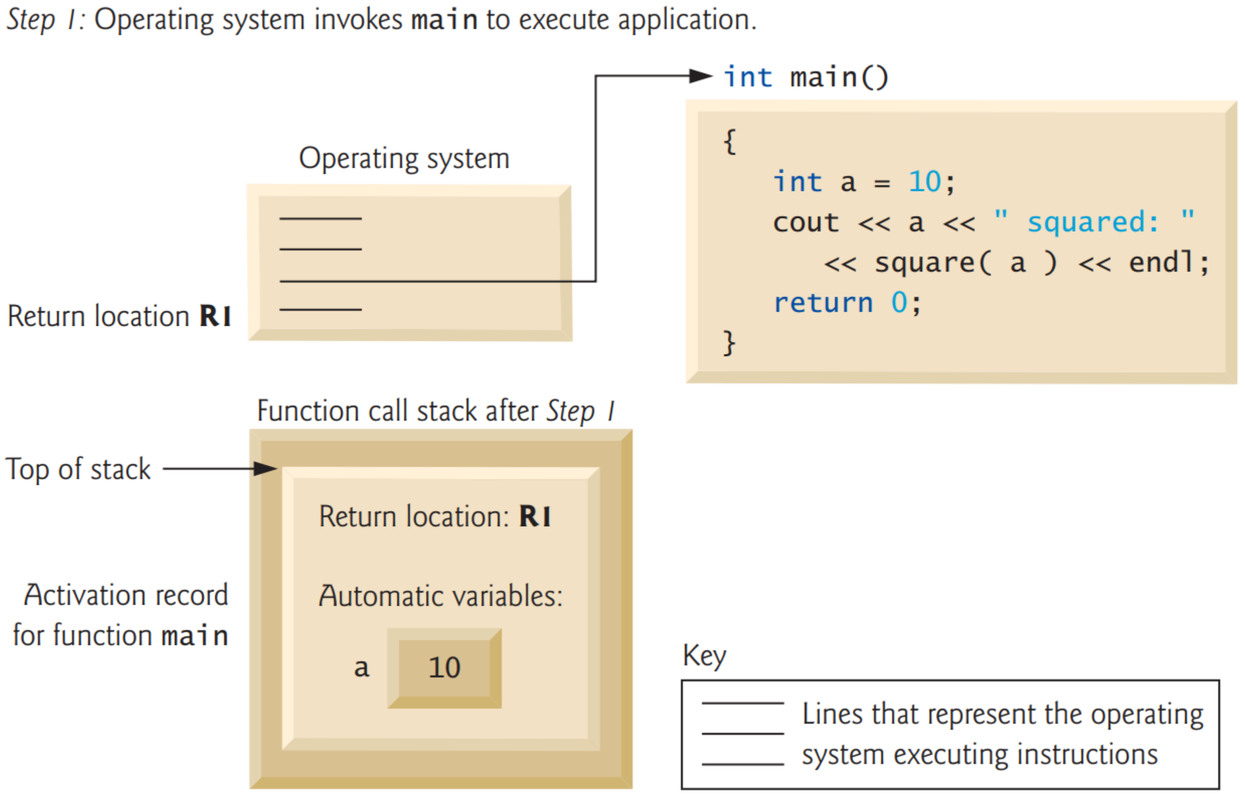
```
#include <iostream>
using namespace std;

int square(int); // prototype for function square

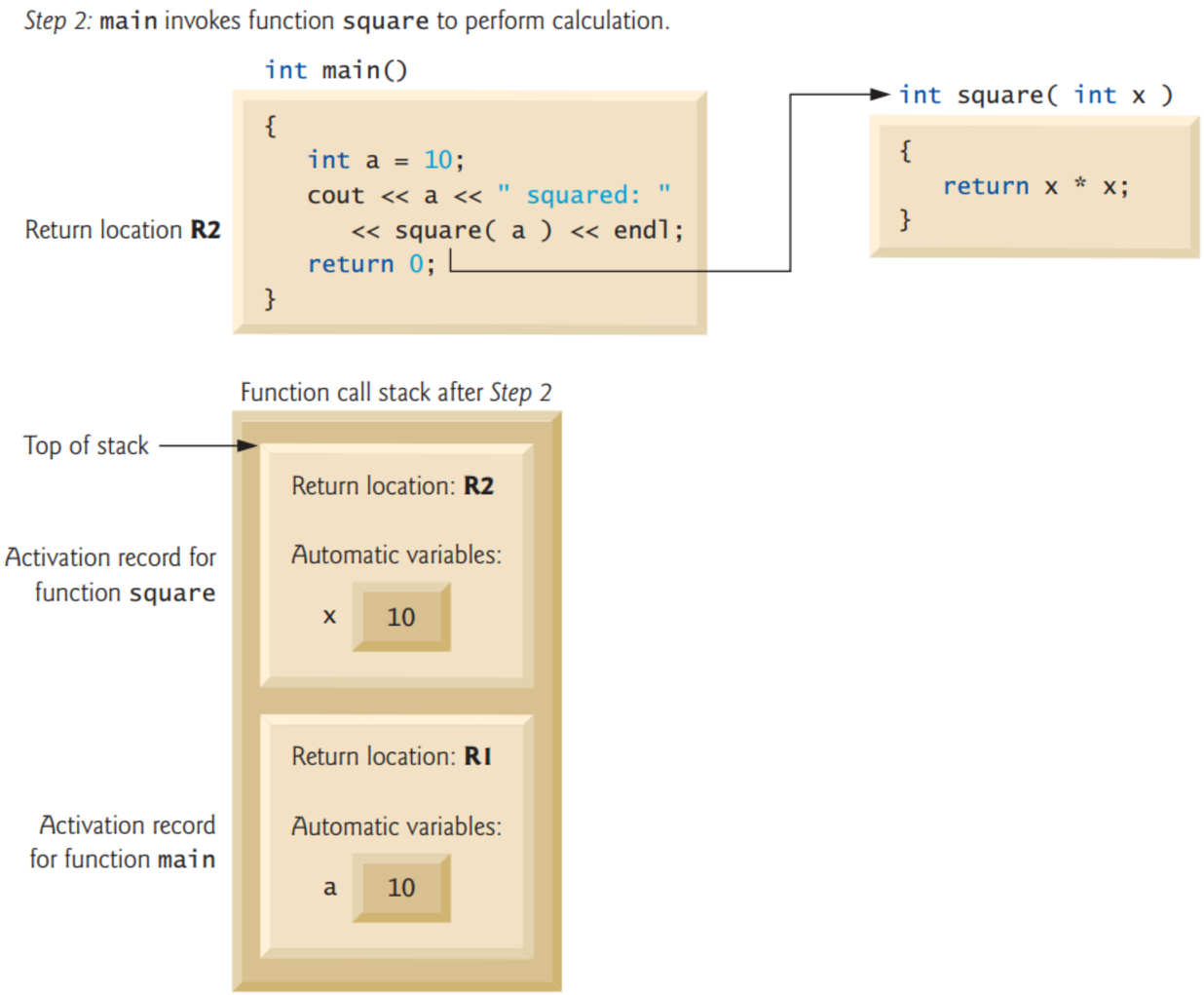
int main() {
   int a = 10;
   cout << a << " squared: " << square(a) << endl;
   return 0;
}

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

Function Call

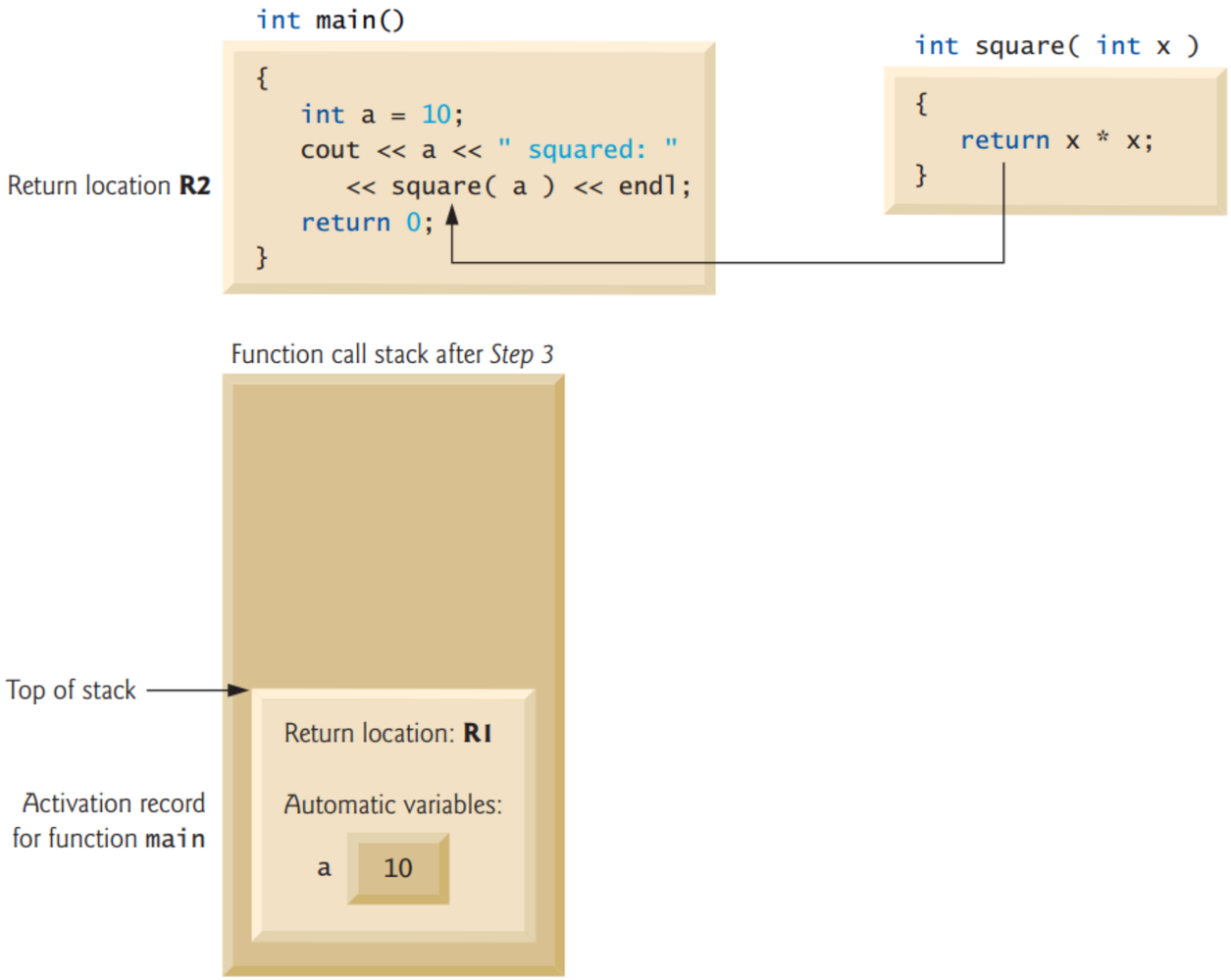


Function Call



Function Call

Step 3: `square` returns its result to `main`.



Storage

- Common errors:
  - Use variables without initialization
  - Memory fault
  - Access restricted areas
  - Memory corruption

Uninitialized variables

```
#include <iostream>
#include <math.h>
using namespace std;

float val;

float foo(float a, float b) {
    val += b;
    return a * b + val;
}

int main() {
    float x, y;
    x = 0.5f;
    cout << foo(x, y) << endl;
    return 0;
}
```

Memory fault (access freed memory)

```
#include <iostream>
#include <math.h>
using namespace std;

float* foo(float a, float b) {
    a += b;
    return &a;
}

int main() {
    float x, y;
    x = 0.5f;
    y = 3.9f;
    float *pRet = foo(x, y);
    cout << *pRet << endl;
    return 0;
}
```

Memory corruption

```
#include <iostream>
#include <math.h>

void foo(char *pStr) {
    char buf[10];
    strcpy_s(buf, pStr);
}

int main() {
    char pStr [] = "This string will overwrite the local buffer";
    foo(pStr);
    return 0;
}
```

Summarise

- Learn about functions: how to define and use in the program
- How to pass parameters, understand scope of variables
- Memory organization of a program