

Chapter 5. Blocks, Functions and Reference Variables

Programming Concepts in Scientific Computing
EPFL, Master class

October 11, 2023

Blocks

Syntax

```
{  
  // SOME CODE  
}
```

Blocks

Syntax

```
{  
  // SOME CODE  
}
```

Blocks

Scope of a variable

```
// Block 1
{
    int i = 5; // local to Block 1
    // Block 2
    {
        int j = 10; // local to Block 2
        i = 10;     // inherited from Block 1
    }
    // variable j is destructed
    j = 5; // so ?
}
// variable i is destructed
```

Blocks

Scope of a variable

```
// Block 1
{
    int i = 5; // local to Block 1
    // Block 2
    {
        int j = 10; // local to Block 2
        i = 10;     // inherited from Block 1
    }
    // variable j is destructed
    j = 5; // so ?
}
// variable i is destructed
```

Blocks

Scope of a variable

```
// Block 1
{
    int i = 5; // local to Block 1
    // Block 2
    {
        int j = 10; // local to Block 2
        i = 10;     // inherited from Block 1
    }
    // variable j is destructed
    j = 5; // so ?
}
// variable i is destructed
```

Blocks

Local vs global variables

```
int i = 5; // global variable

int main() {
    int j = 7; // local variable
    std::cout << i << "\n";
    {
        int i = 10, j = 11;
        std::cout << i << "\n";    // local value of i is 10
        std::cout << ::i << "\n"; // global value of i is 5
        std::cout << j << "\n";    // value of j here is 11
    }
    std::cout << j << "\n"; // value of j here is 7
    return 0;
}
```

Blocks

Local vs global variables

```
int i = 5; // global variable
```

```
int main() {  
    int j = 7; // local variable  
    std::cout << i << "\n";  
    {  
        int i = 10, j = 11;  
        std::cout << i << "\n";    // local value of i is 10  
        std::cout << ::i << "\n"; // global value of i is 5  
        std::cout << j << "\n";    // value of j here is 11  
    }  
    std::cout << j << "\n"; // value of j here is 7  
    return 0;  
}
```


Blocks

Local vs global variables

```
int i = 5; // global variable
```

```
int main() {  
    int j = 7; // local variable  
    std::cout << i << "\n";  
    {  
        int i = 10, j = 11;  
        std::cout << i << "\n";    // local value of i is 10  
        std::cout << ::i << "\n"; // global value of i is 5  
        std::cout << j << "\n";    // value of j here is 11  
    }  
    std::cout << j << "\n"; // value of j here is 7  
    return 0;  
}
```

Blocks

Local vs global variables

```
int i = 5; // global variable

int main() {
    int j = 7; // local variable
    std::cout << i << "\n";
    {
        int i = 10, j = 11;
        std::cout << i << "\n"; //
        std::cout << ::i << "\n"; // global value of i is 5
        std::cout << j << "\n"; // value of j here is 11
    }
    std::cout << j << "\n"; // value of j here is 7
    return 0;
}
```

Blocks

Local vs global variables

```
int i = 5; // global variable

int main() {
    int j = 7; // local variable
    std::cout << i << "\n";
    {
        int i = 10, j = 11;
        std::cout << i << "\n";    // local value of i is 10
        std::cout << ::i << "\n"; // global value of i is 5
        std::cout << j << "\n";    // local value of j is 11
    }
    std::cout << j << "\n"; // value of j here is 7
    return 0;
}
```

Blocks

Local vs global variables

```
int i = 5; // global variable

int main() {
    int j = 7; // local variable
    std::cout << i << "\n";
    {
        int i = 10, j = 11;
        std::cout << i << "\n";    // local value of i is 10
        std::cout << ::i << "\n"; // global value of i is 5
        std::cout << j << "\n";    // value of j here is 11
    }
    std::cout << j << "\n"; // value of j here is 7
    return 0;
}
```

Blocks

Local vs global variables

```
namespace PCSC {  
  
    int i = 5; // global variable  
}  
  
int main() {  
    std::cout << PCSC::i;  
    std::cout << std::endl;  
}
```

Blocks

Local vs global variables

```
namespace PCSC {
```

```
    int i = 5; // global variable  
}
```

```
int main() {  
    std::cout << PCSC::i;  
    std::cout << std::endl;  
}
```

Blocks

Local vs global variables

```
namespace PCSC {  
  
    int i = 5; // global variable  
}  
  
int main() {  
    std::cout << PCSC::i;  
    std::cout << std::endl;  
}
```

Functions

Simple Functions

Declaration/Prototype:

```
double CalculateMinimum(double x, double y);
```

- ▶ Function name
- ▶ Return type
- ▶ Typed parameters

Functions

Simple Functions

Declaration/Prototype:

```
double CalculateMinimum(double x, double y)
```

- ▶ Function name
- ▶ Return type
- ▶ Typed parameters

Functions

Simple Functions

Declaration:

```
double CalculateMinimum(double x, double y);
```

Functions

Simple Functions

Declaration:

```
double CalculateMinimum(double x, double y);
```

Usage:

```
double x = 4.0, y = -8.0;  
double minimum_value = CalculateMinimum(x, y);  
std::cout << "min = " << minimum_value << "\n";
```

Functions

Simple Functions

Declaration:

```
double CalculateMinimum(double x, double y);
```

Usage:

```
double x = 4.0, y = -8.0;
double minimum_value = CalculateMinimum(x, y);
std::cout << "min = " << minimum_value << "\n";
```

Definition:

```
double CalculateMinimum(double a, double b) {
    if (a < b) {
        return a;
    }
    return b;
}
```

Concept of interfaces

- ▶ Describes how to use a function

Concept of interfaces

- ▶ Describes how to use a function
- ▶ Opposed to the function body: implementation

Concept of interfaces

- ▶ Describes how to use a function
- ▶ Opposed to the function body: implementation
- ▶ An interface is usually written in .hh/.hpp (header) files

Concept of interfaces

- ▶ Describes how to use a function
- ▶ Opposed to the function body: implementation
- ▶ An interface is usually written in .hh/.hpp (header) files

Why is it important to have this concept ?

Concept of interfaces

- ▶ Describes how to use a function
- ▶ Opposed to the function body: implementation
- ▶ An interface is usually written in .hh/.hpp (header) files

Why is it important to have this concept ?

- ▶ Allow collaborative work
- ▶ Normalizes the knowledge needed to call a function
- ▶ Limits modifications in cascade

Header files

Example: CalculateMinimum.hpp

```
#ifndef CALCULATEMINIMUM_HPP  
#define CALCULATEMINIMUM_HPP  
  
double CalculateMinimum(double a, double b);  
  
#endif
```

Header files

Example: CalculateMinimum.hpp

```
#ifndef CALCULATEMINIMUM_HPP  
#define CALCULATEMINIMUM_HPP
```

```
double CalculateMinimum(double a
```

```
#endif
```

Header files

Example: CalculateMinimum.hpp

```
#ifndef CALCULATEMINIMUM_HPP  
#define CALCULATEMINIMUM_HPP
```

```
double CalculateMinimum(double a, double b);
```

```
#endif
```

Usage

```
#include "calculate_minimum.hpp"
#include <iostream>

int main(int argc, char *argv[]) {

    double x = 4.0, y = -8.0;
    double minimum_value = CalculateMinimum(x, y);
    std::cout << "min = " << minimum_value << "\n";

    return 0;
}
```

Usage

```
#include "calculate_minimum.hpp"
#include <iostream>

int main(int argc, char *argv[]) {

    double x = 4.0, y = -8.0;
    double minimum_value = CalculateMinimum(x,
    std::cout << "min = " << minimum_value << "\n";

    return 0;
}
```

Implementation file

Example: CalculateMinimum.cpp

```
double CalculateMinimum(double a, double b) {  
    if (a < b) {  
        return a;  
    }  
    return b;  
}
```

Functions

Returning an array

Return the pointer to the allocated memory!

```
double *allocateVector(int size) {  
    double *v = new double[size];  
    return v;  
}
```


Functions

Returning a Matrix

Return the pointer to the allocated memory!

```
double **allocateMatrix(int m, int n) {  
  
    double **mat = new double *[m];  
    for (int i = 0; i < m; ++i) {  
        mat[i] = new double[n];  
    }  
    return mat;  
}
```

Functions

Input with pointer

```
void assign_by_value(double value) { value = 10; }
```

```
void assign_by_pointer(double *value) { *value = 10; }
```

What is the difference ?

Functions

Input with pointer

```
void assign_by_value(double value) { value = 10; }
```

```
void assign_by_pointer(double *value) { *value = 10; }
```

What is the difference ?

- ▶ The difference is the scope (life duration) of the variable value
- ▶ Pointer argument allows to change the pointed value
- ▶ Non-Pointer arguments are simply copied

Functions

Array Input

```
double doIt(double array[]) {  
    array[1] = 10.;  
    return array[1];  
}  
  
double u[10];  
std::cout << doIt(u) << std::endl;  
double *u2 = new double[10];  
std::cout << doIt(u2) << std::endl;
```

Functions

Array Input

```
double doIt(double *array) {  
    array[1] = 10.;  
    return array[1];  
}  
  
double u[10];  
std::cout << doIt(u) << std::endl;  
double *u2 = new double[10];  
std::cout << doIt(u2) << std::endl;
```

Functions

Default parameter value

```
double doIt(double a, double b = 0.) { return a + b; }
```

Functions

Default parameter value

```
double doIt(double a, double b = 0.) { return a + b; }
```

```
std::cout << doIt(10., 5.) << std::endl;
```

Functions

Default parameter value

```
double doIt(double a, double b = 0.) { return a + b; }
```

```
std::cout << doIt(10., 5.) << std::endl;
```

```
std::cout << doIt(10.) << std::endl;
```


Functions

Polymorphism/Overloading

Several functions with the same name:

- ▶ They **MUST** be distinguishable by their arguments(number and types) and return type

Functions

Polymorphism/Overloading

Several functions with the same name:

- ▶ They **MUST** be distinguishable by their arguments(number and types) and return type

This is possible

```
double doIt(double a, double b);  
double doIt(int a, int b = 0);
```

This is not

Functions

Polymorphism/Overloading

Several functions with the same name:

- ▶ They **MUST** be distinguishable by their arguments(number and types) and return type

This is possible

```
double doIt(double a, double b);  
double doIt(int a, int b = 0);
```

This is not

```
double doIt(double a);  
int doIt(double a); // not compiling
```

Functions

Polymorphism/Overloading

Several functions with the same name:

- ▶ They **MUST** be distinguishable by their arguments(number and types) and return type

This is possible

```
double doIt(double a, double b);  
double doIt(int a, int b = 0);
```

This is not

```
double doIt(double a);  
int doIt(double a); // not compiling  
  
int doIt(int a, int b = 0);  
int doIt(int a); // not usable
```

Recursive Functions

Function which calls itself inside the body.

```
int factorial(int a) {  
    if (a == 1)  
        return 1;  
    return a * factorial(a - 1);  
}
```

Recursive Functions

```
int factorial(int a) {  
    if (a == 1)  
        return 1;  
    return a * factorial(a - 1);  
}
```

```
int main() {  
    int a = factorial(6);  
    return 0;  
}
```

Functions

Pointer to function

The function

```
double foo(double a) { return a + 1; }
```

Functions

Pointer to function

The function

```
double foo(double a) { return a + 1; }
```

The pointer

```
double (*ptr_foo)(double a) = &foo;
```


Functions

Pointer to function

The function

```
double foo(double a) { return a + 1; }
```

The pointer

```
double (*ptr_foo)(double a) = &foo;
```

```
std::function<double(double a)> ptr_foo_modern = &foo;
```

Functions

Pointer to function

The function

```
double foo(double a) { return a + 1; }
```

The pointer

```
double (*ptr_foo)(double a) = &foo;
```

```
std::function<double(double a)> ptr_foo_modern = &foo;
```

Standard library, clearer

The function call

```
ptr_foo(10);  
ptr_foo_modern(10);
```

References

A *practical* syntax of C++: the references

```
void foo(double &a) { a = 10.; }
```

What is the difference between pointers and references ?

References

```
int main() {  
    int a = 1;  
    int &b = a;  
    int &c = a;  
    int &d = a;  
}
```

(gdb) x/20xw &a

References

```
int a = 1, b = 2;  
int *ptr = &a;  
ptr = &b;  
int &ref = a;
```

References

```
int a = 1, b = 2;  
int *ptr = &a;  
ptr = &b;  
int &ref = a;
```

- ▶ The usage: you don't need to use the '*' operator
- ▶ A reference points to a value that is 'read only'
- ▶ Not possible to change where the reference points to
- ▶ Not possible to increment the internal pointer

References: example of usage

Can do the following:

Return a reference (an alias of variable)

```
double &getValue(double *A, int ncols, int nrows, //  
                int i, int j) {  
    return A[i * ncols + j];  
}
```

Allowing:

```
void foo_ptr(const double *ptr) {
```

References: example of usage

What are the differences between these:

```
void foo_ref(const double &ref) {  
    std::cout << ref;  
    // ref = 2; // can do ?  
}
```

```
void foo_val(double val) {  
    std::cout << val;  
    // val = 2; // can do ?  
}
```

```
void foo_ptr(const double *ptr) {  
    std::cout << ptr;  
    // *ptr = 2; // can do ?  
}
```


Right and Left References

With the function:

```
void print(double &a) { std::cout << a; }
```

You can do:

```
double a = 10;  
print(a);
```

But can you do:

```
print(100);
```

Right and Left References

A possibility is:

```
void print(const double &a) { std::cout << a; }
```

Q: Why const change the situation?

Or the **Right reference (R-reference)**:

```
void print2(double &&a) { std::cout << a; }
```

Right and Left References

The name **Right** and/or **Left** references comes from an assignment statement:

$$\underbrace{A}_{\text{left}} = \underbrace{B + C}_{\text{right}};$$

Right and Left References

The name **Right** and/or **Left** references comes from an assignment statement:

$$\underbrace{A}_{\text{left}} = \underbrace{B + C}_{\text{right}};$$

- ▶ Left side: needs memory storage to be decided (by compiler)
- ▶ Right side: memory storage is “temporary”

Most common usage is:

```
// right reference combined with auto   Auto keyword  
auto &&a = 100;  
auto &&b = a;
```

Blocks, Functions, References

Take away message

- ▶ **block/scope**: defines the life duration of (static) variables
- ▶ **namespace**: possibility to name a block (can also be nested)
- ▶ **interface concept**: Distinguish *how to use* from *implementation*
- ▶ **Parameters of functions**: passed by copy, by pointer or by reference
- ▶ **Parameters default**: example
`void foo(int a, double b=1.5);`
- ▶ **Overloading**: different functions may have the same name (prototype/signature must be different)
- ▶ **References**: Convenient read only pointer
- ▶ **Left/right References**: (Advanced) Convey temporariness to compiler