

©Bennet Becker, 2019

IP of the Max-Planck-Institute for Physics of Complex Systems (mpipks),
Nöthnitzer Straße 38, 01187 Dresden

DO NOT USE OUTSIDE THE MPI PKS

Images Copyrighted by their owners

Theme Metropolis CC-BY-SA 4.0 by github.com/matze



MPI PKS C++ Lecture

Bennet Becker

2019-09-09

MPI PKS

Program Structure

Control Structure

Control Structure i

- a simple statement is each individual instruction of a program. like expressions seen before.
- statements are always terminated with a semicolon (most common error source), and executed in the same order in which they appear in a program*.
- not limited to such linear sequences of statements
- → control-flow statements. they require a generic (sub)statement as part of its syntax. single statements, terminated with semicolon or compound-statement, enclosed with curly brackets



Control Structure ii

```
// single statement
statement;

// compound statement
{
    statement1;
    statement2;
    ...
    statementN;
}
```

Selection statements - if and else i

```
// simple version: single statement
if (condition) statement;

if (condition)
    statement1;
else
    statement2;
```



Selection statements - if and else ii

```
// with compound statement and alternatives
if (condition) {
    statement1;
    statement2;
    ...
    statementN;
}
```


Selection statements - if and else iii

```
if (condition) {  
    statement11;  
    statement12;  
    ...  
    statement1N;  
} else {  
    statement21;  
    statement22;  
    ...  
    statement2N;  
}
```

Selection statements - if and else iv

```
if (condition) {  
    statement11;  
    statement12;  
    ...  
    statement1N;  
} else if (condition2) {  
    statement21;  
    statement22;  
    ...  
    statement2N;  
} else {  
    statement31;  
    statement32;  
    ...  
    statement3N;  
}
```



Selection statements - switch

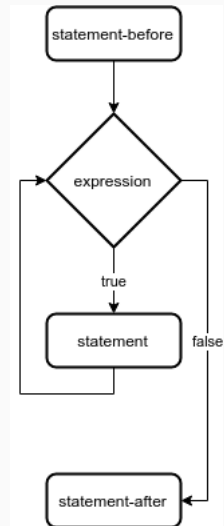
```
// switch with many multiple options
switch (varibale) {
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
    ...
    case valueN:
        statementN;
        break;
    default:
        statement;
}
```

break is important. because switch jumps to according `case value:` and executes all statement until end or next break (which can be useful if used intentionally).



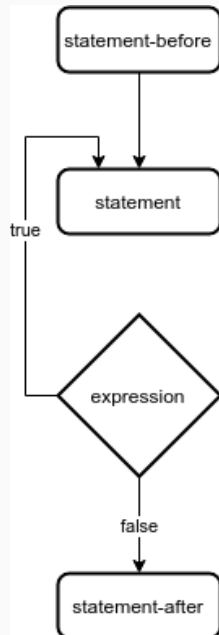
Iteration statements - while

```
while (expression) {  
    statements;  
}
```



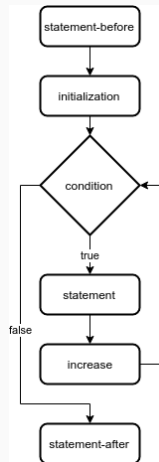
Iteration statements - do-while

```
do {  
    statements;  
} while (expression);
```



Iteration statements - for loop

```
for (initialization; condition;  
    ↪ increase) {  
    statement;  
}
```



Iteration statements - Range based for loop

- `string str; for (char c : str){ }`
- `int arr[]; for(int i : arr){ }`
- `for (auto i : var){ }`



Jump statements i

- `break` : break/abort a loop even if continuation condition is still fulfilled
- `continue` : skip rest of this loop iteration and continue with next iteration
- `goto` : **NON EXISTENT!**



Jump statements ii



Jump statements iii

Go To Statement Considered Harmful
Go To Statement Considered Harmful
Go To Statement Considered Harmful
Go To Statement Considered Harmful
Go To Statement Considered Harmful
Go To Statement Considered Harmful
Go To Statement Considered Harmful
Go To Statement Considered Harmful
Go To Statement Considered Harmful
Go To Statement Considered Harmful
Go To Statement Considered Harmful



TASK: Greatest common divisor

Write a program, that for two positive Numbers, calculates thier Greatest common divisor.

Begin like this

```
#include <iostream>
#include <cstdlib>

int main(int argc, const char** argv){
    int n1, n2;

    std::cout << "Enter two numbers: ";
    std::cin >> n1 >> n2;

    //... your code here...

    return 0;
}
```

Possible solutions

```
#include <iostream>
#include <cstdlib>

int main(int argc, const char** argv){
    int n1, n2;

    std::cout << "Enter two numbers: ";
    std::cin >> n1 >> n2;

    while(n1 != n2) {
        if(n1 > n2) {
            n1 -= n2;
        } else {
            n2 -= n1;
        }
    }

    std::cout << "GCD = " << n1;
    return 0;
}
```



Possible solutions

```
#include <iostream>
#include <cstdlib>

#define ABS(x) x < 0 ? x * -1 : x

int main(int argc, const char** argv){
    int a, b, h;

    std::cout << "Enter two numbers: ";
    std::cin >> a >> b;

    if (a == 0){
        std::cout << "GCD = " << ABS(b);
        return 0;
    }
    if (b == 0) {
        std::cout << "GCD = " << ABS(a);
        return 0;
    }

    do {
        h = a % b;
        a = b;
        b = h;
    } while (b != 0);

    std::cout << "GCD = " << ABS(a);
    return 0;
}
```



Possible solutions

```
#include <iostream>
#define EVEN(x) (x % 2) == 0

int main(int argc, const char** argv){
    int a,b;
    int d;

    std::cout << "Enter two numbers: ";
    std::cin >> a >> b;

    while(EVEN(a) && EVEN(b)){
        a >>= 1; // a = a / 2
        b >>= 1; // b = b / 2
        d++;
    }
    while(a != b){
        if(EVEN(a)){
            a >>= 1;
        } else if (EVEN(b)){
            b >>= 1;
        } else if (a > b) {
            a = (a - b) >> 1;
        } else {
            b = (b - a) >> 1;
        }
    }

    std::cout << (a << d); // a * 2 ^ d
    return 0;
}
```



Functions

Functions i

- Functions allow to structure programs in segments of code to perform individual tasks

```
1      type_t1 func_name(type_t2 param1, type_t3 param2,  
      ↪ ...){  
2      type_t1 val;  
3      // ...  
4  
5      return val;  
6      }
```


Functions ii

- `type_t1` is the type of the value returned by the function
- `func_name` identifier by which the function is called
- `param1` - `paramN` : parameters (with type). as many as needed
- Functions with no type. -> `void function(type parm1, ...) { }`
- Function with no parameters -> `type function(void) { }` or `type function() { }`
- Default parameters ->
`type function(type1 param1, type2 param2 = value) { }`
(parameter `param2` can now be omitted on function-call)
- parentheses required on function call! to differentiate a function from a variable



main() - Function

- return value of `main()`
 - `main` returns `int`
 - value `0` is interpreted by the environment as that the program ended successfully
 - but `main` may also return other values. some environments give access to that value to the caller in some way, although this behavior is not required nor necessarily portable between platforms
 - The values for `main` that are guaranteed to be interpreted in the same way on all platforms are:
 - `0` : The program was successful
 - `EXIT_SUCCESS` : The program was successful (same as above). This value is defined in header `#include <stdlib>` .
 - `EXIT_FAILURE` : The program failed. This value is defined in header `#include <stdlib>` .



- Declaring functions by writing them without body:

```
type_t1 func_name(type_t2 param1, type_t3 param2, ...);
```

- functions as well as variables can not be used before declared
- So you can declare functions before defining them to overcome scoping difficulties



TASK: least-common-multiple

Rewrite the GCD-Program to use a function. Use this function to calculate the least-common-multiple.

```
#include <iostream>

// ...

int main (int argc, const char** argv){
    int a,b;

    std::cout << "Enter two numbers: ";
    std::cin >> a >> b;

    std::cout << "GCD = " << /* ... */ << std::endl
              << "LCM = " << /* ... */ << std::endl;
}

// ...
```

Possible Solution i

```
#include <iostream>

int gcd(int, int);
int lcm(int, int);

int main (int argc, const char** argv){
    int a,b;

    std::cout << "Enter two numbers: ";
    std::cin >> a >> b;

    std::cout << "GCD = " << gcd(a,b) << std::endl
              << "LCM = " << lcm(a,b) << std::endl;
}
```



Possible Solution ii

```
int gcd(int a, int b){  
    if (a == 0){  
        return std::abs(b);  
    }  
    if (b == 0) {  
        return std::abs(a);  
    }  
  
    do {  
        h = a % b;  
        a = b;  
        b = h;  
    } while (b != 0);  
  
    return std::abs(a);  
}  
  
int lcm(int a, int b){  
    return std::abs(a * b / gcd(a,b));  
}
```

- above functions are called by value. Copying the value of arguments to the functions parameters
- calling a function by reference, passes the references (addresses) of the variable used as parameters to the function. The function can then modify the values directly

Call-by-reference ii

```
1  // passing parameters by reference
2  #include <iostream>
3
4  void duplicate (int& a, int& b, int& c)
5  {
6      a *= 2;
7      b *= 2;
8      c *= 2;
9  }
10
11 int main (int argc, const char** argv)
12 {
13     int x=1, y=3, z=7;
14     duplicate (x, y, z);
15     std::cout << "x=" << x << ", y=" << y << ", z=" << z;
16     return 0;
17 }
```

- the ampersands in line 4 mean that it is a reference to an integer



- References are type-safe and can be considered as an alternate name for a variable. they behave similar to pointers but are much more restricted and much less powerful

Call-by-reference - Efficiency i

- copying values of fundamental types such as int is relatively inexpensive. But if the parameter is of a large compound type, it may result on certain overhead
- E.g. instead of passing large strings

```
string concatenate (string a, string b)
{
    return a+b;
}
```

only pass references



Call-by-reference - Efficiency ii

```
string concatenate (string& a, string& b)
{
    return a+b;
}
```

This is faster and with much less memory overhead

- Drawback: the function is now able to modify the value. Solution define parameters as constant

Call-by-reference - Efficiency iii

```
string concatenate (const string& a, const string& b)
{
    return a+b;
}
```

Now the function is forbidden to modify the parameter values

Function inlining i

- calling a function involves a certain overhead (stacking arguments, jumps, etc...)
- so for very short functions it might be more efficient to insert the code directly where the function is called
- but doing this manually would introduce redundancy and makes it harder to change the function when necessary
- you can inform your compiler to do it for you. with the keyword `inline`



Function inlining ii

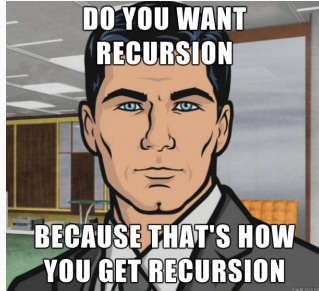
```
inline string concatenate (const string& a, const string&  
    ↪ b)  
{  
    return a+b;  
}
```

- Note that most modern compilers already optimize code to generate inline functions when they see an opportunity to improve efficiency, even if not explicitly marked with the inline specifier.
- Therefore, this specifier merely indicates the compiler that inline is preferred for this function (although the compiler is free to not inline it, and optimize otherwise).



Recursion

see Recursion



Recursion

- Recursivity is the property that functions have to be called by themselves
- Useful for some tasks:
 - quicksort
 - fibonacci
 - factorial
- if the recursive function call is the last thing in the function, the function is called tail-recursive
 - better optimizeable than non-tail-recursive functions



TASK: Fibonacci and Factorial

Implement the factorial and fibonacci recursive and iterative. Compare runtime.
Can we get even faster?



overloads and templates i

- two functions can have the same name if their parameters are different
- because of different number of parameters or type of parameters is different
- two functions with same name, only differing in return-type are **not** possible

```
#include <iostream>

int operate (int a, int b)
{
    return (a*b);
}

double operate (double a, double b)
{
    return (a/b);
}
```

overloads and templates ii

- this is called overloading
- Overloaded functions may have the same definition

```
// overloaded functions
#include <iostream>

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

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

- could be overloaded for a lot of types, and it could make sense for all of them to have the same body



overloads and templates iii

- C++ has the ability to define functions with generic types, known as function templates
- `template <templ-param> function-declaration`

```
template <typename T>
T sum (T a, T b) {
    return a+b;
}
```

- `T` is a generic type in this functions and can be used like any other type
- call the function

```
name <template-arguments> (function-arguments)
```



overloads and templates iv

- `x = sum<int>(10,20);`
- templates are not limited to type names

```
#include <iostream>

template <class T, int N>
T fixed_multiply (T val)
{
    return val * N;
}

int main() {
    std::cout << fixed_multiply<int,2>(10) << '\n';
    std::cout << fixed_multiply<int,3>(10) << '\n';
}
```

- The second argument of the `fixed_multiply` function template is of type `int`. It just looks like a regular function parameter, and can actually be used just like one.



- **But:** the value of template parameters is determined on compile-time to generate a different instantiation of the function and thus the value of that argument is never passed during runtime. Therefore the integer template argument needs to be a constant expression

ie, the argument of the template (the type of the function) cannot be a variable, it needs to be literally written when calling the template. It's equivalent to define a new function.

Scoping i




- Named entities, such as variables, functions, and compound types need to be declared before being used in C++
- An entity declared outside any block has *global scope*, meaning that its name is valid anywhere in the (following) code.
- While an entity declared within a block (such as a function or a selective statement), has *block scope*, and is only visible within the specific block in which it is declared, but not outside it.
- Variables with block scope are known as *local variables*
- In each scope, a name can only represent one entity. There cannot be two variables with the same name in the same scope



- The visibility of an entity with *block scope* extends until the end of the block, including inner blocks
- inner block, can re-utilize a name existing in an outer scope to refer to a different entity; in this case, the name will refer to a different entity only within the inner block, hiding the entity it names outside

Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a					0
void f					—
int parameter					as passed
float a					3.14
void g					—
double par1					as passed
int everything					42
int main					—
int argc					
int argv					
int counter					??



Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21				0
void f					—
int parameter					as passed
float a					3.14
void g					—
double par1					as passed
int everything					42
int main					—
int argc					
int argv					
int counter					??



Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21				0
void f	6-21				—
int parameter					as passed
float a					3.14
void g					—
double par1					as passed
int everything					42
int main					—
int argc					
int argv					
int counter					??



Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21				0
void f	6-21				—
int parameter	6-8				as passed
float a					3.14
void g					—
double par1					as passed
int everything					42
int main					—
int argc					
int argv					
int counter					??



Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21				0
void f	6-21				—
int parameter	6-8				as passed
float a	7-8				3.14
void g					—
double par1					as passed
int everything					42
int main					—
int argc					
int argv					
int counter					??



Visibility of Variables (Scopes)

```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```




name	visible	 f	 g	 main	value
int a	4-21				0
void f	6-21				—
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				—
double par1					as passed
int everything					42
int main					—
int argc					
int argv					
int counter					??



Visibility of Variables (Scopes)

```




1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```

name	visible	 f	 g	 main	value
int a	4-21				0
void f	6-21				–
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				–
double par1	10-12				as passed
int everything					42
int main					–
int argc					
int argv					
int counter					??



Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21				0
void f	6-21				—
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				—
double par1	10-12				as passed
int everything	11-12				42
int main					—
int argc					
int argv					
int counter					??



Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21				0
void f	6-21				–
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc					
int argv					
int counter					??



Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21				0
void f	6-21				—
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				—
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				—
int argc	14-21				
int argv					
int counter					??



Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21				0
void f	6-21				–
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter					??



Visibility of Variables (Scopes)

```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21				0
void f	6-21				–
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }




```

name	visible	 f	 g	 main	value
int a	4-21	✗			0
void f	6-21				–
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible				value
		f	g	main	
int a	4-21	✗	✓		0
void f	6-21				—
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				—
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				—
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21				—
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				—
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				—
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓			–
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓		–
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```




1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8				as passed
float a	7-8				3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓			as passed
float a	7-8				3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)




```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗		as passed
float a	7-8				3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8				3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }

```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓			3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗		3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21				–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }

```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗			–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }

```

name	visible	 f	 g	 main	value
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓		–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }

```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12				as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }

```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗			as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }

```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓		as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12				42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗			42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }

```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓		42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }

```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓		42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }

```

name	visible	 f	 g	 main	value
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21				–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }




```

name	visible	 f	 g	 main	value
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗			–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```

name	visible	 f	 g	 main	value
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗	✗		–
int argc	14-21				
int argv	15-21				
int counter	16-21				??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }

```




name	visible	 f	 g	 main	value
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗	✗	✓	–
int argc	14-21				
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗	✗	✓	–
int argc	14-21	✗			
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗	✗	✓	–
int argc	14-21	✗	✗		
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```




1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗	✗	✓	–
int argc	14-21	✗	✗	✓	
int argv	15-21				
int counter	16-21				??



Visibility of Variables (Scopes)

```
1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗	✗	✓	–
int argc	14-21	✗	✗	✓	
int argv	15-21	✗			
int counter	16-21				??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```




name	visible	 f	 g	 main	value
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗	✗	✓	–
int argc	14-21	✗	✗	✓	
int argv	15-21	✗	✗		
int counter	16-21				??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗	✗	✓	–
int argc	14-21	✗	✗	✓	
int argv	15-21	✗	✗	✓	
int counter	16-21				??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗	✗	✓	–
int argc	14-21	✗	✗	✓	
int argv	15-21	✗	✗	✓	
int counter	16-21	✗			??






Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }

```




name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗	✗	✓	–
int argc	14-21	✗	✗	✓	
int argv	15-21	✗	✗	✓	
int counter	16-21	✗	✗		??



Visibility of Variables (Scopes)

```

1  #include <iostream>
2  #include <cstdlib>
3
4  int a;
5
6  void f(int parameter){
7      float a = 3.14;
8  }
9
10 void g(double par1){
11     int everything = 42;
12 }
13
14 int main(int argc,
15           const char** argv){
16     int counter;
17
18     myfunction(a);
19
20     return EXIT_SUCCESS;
21 }
    
```

name	visible				value
		f	g	main	
int a	4-21	✗	✓	✓	0
void f	6-21	✓	✓	✓	–
int parameter	6-8	✓	✗	✗	as passed
float a	7-8	✓	✗	✗	3.14
void g	10-21	✗	✓	✓	–
double par1	10-12	✗	✓	✗	as passed
int everything	11-12	✗	✓	✗	42
int main	14-21	✗	✗	✓	–
int argc	14-21	✗	✗	✓	
int argv	15-21	✗	✗	✓	
int counter	16-21	✗	✗	✓	??



Namespaces i

- Only one entity can exist with a particular name in a particular scope. Seldom a problem for local names; blocks are short
- non-local names bring more possibilities for name collision, especially considering that libraries may declare many functions, types, and variables, neither of them local in nature, and some of them very generic
- Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them *namespace scope*. Allowing organizing the elements of programs into different logical scopes referred to by names



Namespaces ii

- syntax to declare a namespaces

```
namespace identifier
{
    named_entities
}
```

Where `identifier` is any valid identifier and `named_entities` is the set of variables, types and functions that are included within the namespace.



Namespaces iii

```
// namespaces
#include <iostream>

namespace foo
{
    int value() { return 5; }
}

namespace bar
{
    const double pi = 3.1416;
    double value() { return 2*pi; }
}

int main () {
    std::cout << foo::value() << '\n';
    std::cout << bar::value() << '\n';
    std::cout << bar::pi << '\n';
    return 0;
}
```



Namespaces iv

- Namespace-members are accessed via `::` operator -> `std` is also a Namespace from the C++ Standard Library
- when utilizing libraries like C++ STL always specifying the Namespace can be tedious -> keyword `using` to import Names or whole Namespaces

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

namespace first
{
    int x = 5;
    int y = 10;
}

namespace second
{
    double x = 3.1416;
    double y = 2.7183;
}
```



Namespaces v

```
int a () {
    using first::x;
    using second::y;
    cout << x << '\n';
    cout << y << '\n';
    cout << first::y << '\n';
    cout << second::x << '\n';
    return 0;
}

int b () {
    using namespace first;
    cout << x << '\n';
    cout << y << '\n';
    cout << second::x << '\n';
    cout << second::y << '\n';
    return 0;
}
```

- Namespaces can be aliased: `namespace new_name = current_name;`



Storage Classes

- back to variable initialization...
- storage for variables with *global* or *namespace scope* is allocated for the entire duration of the program. Known as *static storage* (BSS - Block started by symbol)
- storage for *local variables* is only available during the block in which they are declared; after that, that same storage may be used for a local variable of some other function, or used otherwise. This is called *automatic storage*
- But there is another substantial difference between variables with *static storage* and variables with *automatic storage*:
 - Variables with *static storage* (such as global variables) that are not explicitly initialized are guaranteed to be automatically initialized with zeroes.
 - Variables with *automatic storage* (such as local variables) that are not explicitly initialized are left uninitialized, and thus have an undetermined value.

