

C/C++ PRIMER

LECTURE 4: FUNCTION ARGUMENTS, MEMORY ALLOCATION, I/O

Fabian Wermelinger

Harvard University

OUTLINE

- Function arguments: pass-by-value, pass-by-reference
- Memory allocation, stack and heap
- Input/Output (I/O)

FUNCTION ARGUMENTS

FUNCTION ARGUMENTS

Recall the function signature of the Mid-Point rule from the last lecture:

```
1 double mid_point(const double a, const double b, const int n); // this is a declaration
```

In function declarations or definitions (where the function *body* is defined), *a*, *b* and *n* are called *parameters*.

- Parameters are just dummy variables that are used inside the function *definition* as placeholders for the *arguments* when the function is called.
- In a *function declaration* you technically do not need the parameter names, only their types. This is also a *valid function declaration*:

```
1 double mid_point(const double, const double, const int); // this is a declaration
```

FUNCTION ARGUMENTS

Later you call the function like this:

```
1 double a = 1.0;
2 double b = 2.0;
3 int n = 10;
4 mid_point(a, b, n); // function call
```

Here **a**, **b** and **n** are called *arguments* to the function `mid_point`.

- **Function parameters:** dummy variables with *symbolic* meaning only.
- **Function arguments:** variables with *values assigned to them outside of the function scope*.

FUNCTION ARGUMENTS

- In C/C++ function arguments are *pass-by-value*.
- Pass-by-value means that the arguments are *evaluated* and the result is assigned to corresponding parameters.
- Evaluated means this:

```
1 double a = 1.0;  
2 double b = 2.0;  
3 int n = 10;  
4 mid_point(a + b, b, n); // the first argument will be the result of a + b
```

- If your function parameters are *user-defined classes* which may encapsulate data, passing them by value to functions can be very expensive because the data needs to be copied (internally) too. (This remark applies to C++ only.)

FUNCTION ARGUMENTS

- If your function has parameters that are *pointers*, the value of the pointer argument (i.e. *its address*) is passed by value.
- The function parameter is *still assigned a private copy of the pointer value*, but because the pointer value is a *reference* (i.e. a memory address) we can modify the data at the memory address from *within the function*.
- In that case we speak of *pass-by-reference*, even though the value of the reference (memory address) is passed by value.

```
1 void f(double *a, double *const b)
2 {
3     *a = 1.0; // assign the value 1.0 to the memory location stored in a.
4     a = b;    // assign the pointer value of b to a (this overwrites the initial
5               // parameter value when we call the function, we can no longer
6               // reach the memory location that a pointed to initially).
7     b = a;    // compile time error! We are not allowed to overwrite a constant
8               // (b in this example is a true pass-by-reference parameter).
9 }
```

FUNCTION ARGUMENTS

Function parameters passed by value are *local* to the function.
Consider the following example:

```
1 int power(int x, int n)
2 {
3     int result = 1;
4     for (int i = 0; i < n; ++i) {
5         result *= x;
6     }
7     return result;
8 }
```

We do not necessarily need the iteration variable *i*:

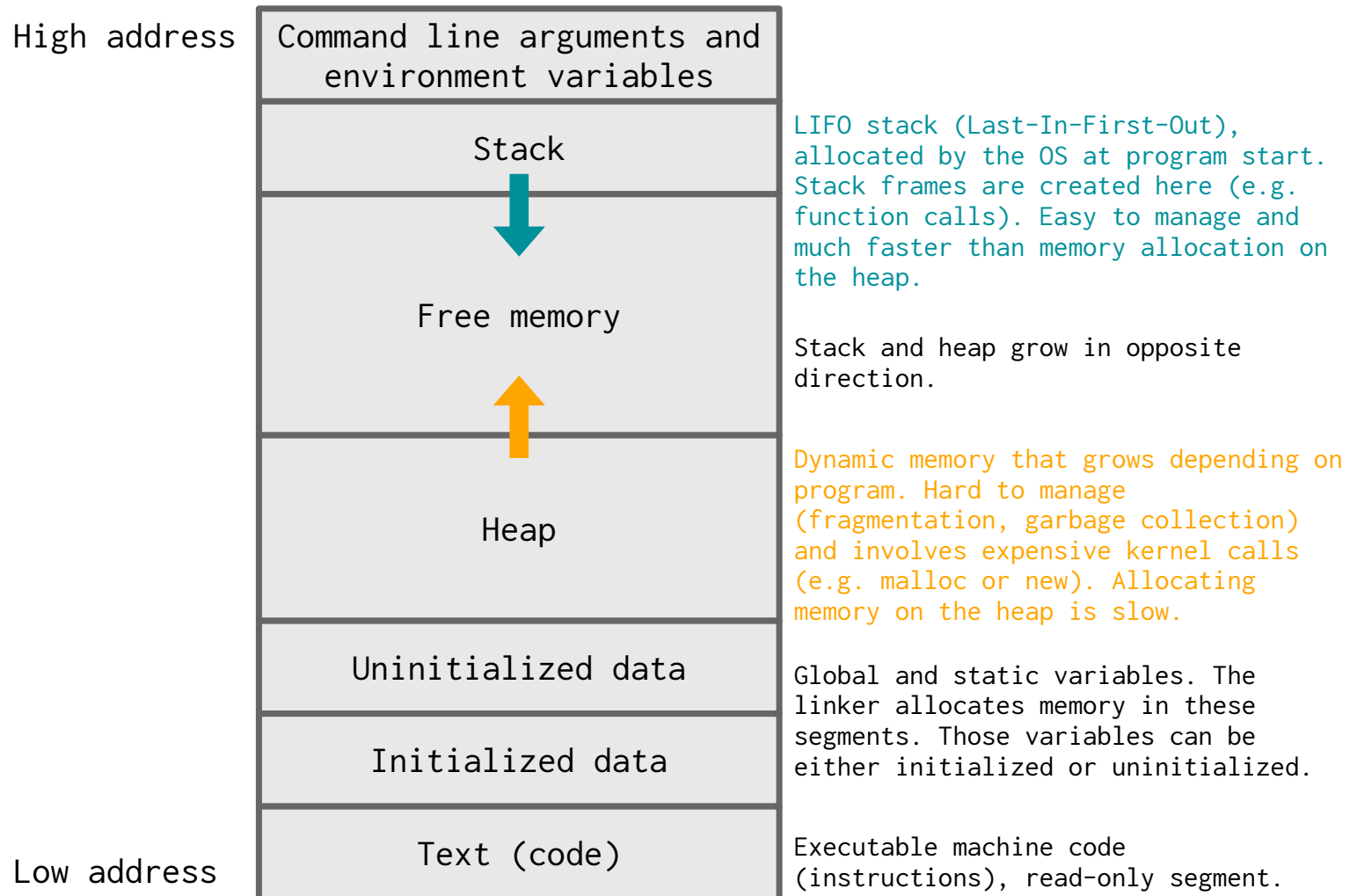
```
1 int power(int x, int n)
2 {
3     int result = 1;
4     do {
5         result *= x;
6     } while (0 < --n); // you can use the parameter n as a normal variable
7     return result;
8 }
```


MEMORY ALLOCATION, STACK AND HEAP

- Your programs require dynamic memory that you allocate on the *heap*.
- Dynamic memory is memory for which you do not know the *size* a priori.
- When you call a function, that function call requires memory too. **Recall:** arguments are pass-by-value, these values must be stored somewhere *temporarily*. The function body itself contains *local* variables that must be allocated somewhere too.
- This memory requirement is much smaller and only of short lifetime. It must therefore be allocated on a data structure that is much faster than the heap. This data structure is called a (LIFO) *stack* (Last-In-First-Out).

MEMORY ALLOCATION, STACK AND HEAP

Structure of virtual memory for an executable program:



MEMORY ALLOCATION, STACK AND HEAP

- The size of the stack is configurable and usually defaults to 4-8 MB.
- On Linux you can check the stack size for programs with `ulimit -s`.
- You can allocate memory on the stack but if you allocate more than what the stack can hold you are creating a ***stack-overflow*** and your program will crash. (This is where the name from the community forum <https://stackoverflow.com> comes from.)

MEMORY ALLOCATION, STACK AND HEAP

Let us create a stack-overflow:

1. Check the stack size on your system:

```
1 $ ulimit -s  
2 8192 # these are kbytes -> 8MB
```

2. Write a small program that allocates at least 8MB on the stack:

```
1 int main(void)  
2 {  
3     double large_array[1048576]; // exactly 8192 kB  
4     return static_cast<int>(large_array[10]);  
5 }
```

3. Compile and run:

```
1 $ g++ stackoverflow.cpp && ./a.out  
2 Segmentation fault (core dumped)
```

- If you use a smaller array the program completes correctly, e.g. `double large_array[1024];`
- Because there are other things allocated on the stack as well, even if we reduce the number of elements in `large_array` only slightly the program would likely still run into a segmentation fault.

MEMORY ALLOCATION, STACK AND HEAP

- If we need a *large chunk of memory*, we must allocate the memory on the *heap*.
- Allocating memory on the heap implies *explicit* memory management! When you ***allocate*** memory, you *must also free* the memory when you no longer need it.
- If you do not free memory and you loose the reference to it, your code is ***leaking memory*** which can have serious implications when you run the program. ***This is a bug.***

MEMORY ALLOCATION, STACK AND HEAP

- Dynamic memory allocation on the heap using the C standard library:

```
1 #include <stdlib.h>
2 int main(void)
3 {
4     // see https://en.cppreference.com/w/c/memory/malloc
5     double *large_array = (double *)malloc(1048576 * sizeof(double));
6     const double retval = large_array[10];
7     free(large_array); // release the memory when no longer needed
8     return (int)retval; // C-style cast from double to int
9 }
```

- Dynamic memory allocation on the heap in C++:

```
1 int main(void)
2 {
3     // see https://en.cppreference.com/w/cpp/language/new
4     double *large_array = new double[1048576];
5     const double retval = large_array[10];
6     delete[] large_array; // release the memory when no longer needed
7     return static_cast<int>(retval); // C++ style cast from double to int
8 }
```

INPUT/OUTPUT ROUTINES

INPUT/OUTPUT ROUTINES

- In C the functions to print formatted output are called `printf` and `fprintf`.
- There are few more but we will not discuss them here. See <https://en.cppreference.com/w/c/io/fprintf> for more documentation.
- We focus here on the C++ I/O routines.

INPUT/OUTPUT ROUTINES

- The standard I/O streams in C++ are defined in the `iostream` header.
- The three main streams are
 - `cin` (stdin)
 - `cout` (stdout)
 - `cerr` (stderr)
- **Example:** read from stdin and print to stdout

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4 int main(void)
5 {
6     cout << "Type something: "; // send something to stdout
7     string received;
8     cin >> received; // read from stdin until newline character
9     cout << "You typed: " << received << endl;
10    return 0;
11 }
```

```
1 $ g++ io.cpp && ./a.out
2 Type something: something
3 You typed: something
```

INPUT/OUTPUT ROUTINES

- Since `stdin`, `stdout` and `stderr` are just three special file streams, all C++ stream objects *behave identical*.
- C++ distinguishes between *input streams* (`istream`) and *output streams* (`ostream`). `cin` is an `istream` object and `cout` and `cerr` are `ostream` objects.
- The `istream` and `ostream` objects associated for regular file streams are defined in the `fstream` header.

INPUT/OUTPUT ROUTINES

Example: print function to stream to different objects:

```
1 #include <fstream>
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 void print(ostream &stream, const string message)
7 {
8     stream << message << endl; // append a newline at the end
9 }
10
11 int main(void)
12 {
13     // create an output stream associated to file 'output.txt'. Note: ofstream
14     // is an output file stream object, it inherits from the ostream object.
15     // Because of this inheritance we can use the 'ostream &' reference in the
16     // print() function above.
17     // See https://www.cplusplus.com/reference/fstream/fstream/
18     ofstream my_file("output.txt");
19     print(my_file, "This message is streamed into my file");
20     print(cout, "This message is streamed to stdout");
21     print(cerr, "This message is streamed to stderr");
22     return 0;
23 }
```

INPUT/OUTPUT ROUTINES

Example: print function to stream to different objects:

```
1 $ g++ print.cpp
2 $ ./a.out > stdout 2> stderr # redirect stdout and stderr streams to files
3 $ cat stdout stderr output.txt
4 This message is streamed to stdout
5 This message is streamed to stderr
6 This message is streamed into my file
```

See also the following link for format manipulators used together with floating point numbers: <https://www.cplusplus.com/reference/ios/scientific/>.
You may find this useful for the following hands-on exercise.

HANDS-ON: DERIVATIVE WITH FINITE-DIFFERENCE

Write a program `dfdx` that computes the derivative of the function

$$f(x) = e^{-\frac{1}{2}x} \sin(x) (\cos(x))^2$$

