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:

```
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*:

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

FUNCTION ARGUMENTS

Later you call the function like this:

```
double a = 1.0;
double b = 2.0;
int n = 10;
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:

```
double a = 1.0;
double b = 2.0;
int n = 10;
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.

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

FUNCTION ARGUMENTS

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

```
int power(int x, int n)
{
    int result = 1;
    for (int i = 0; i < n; ++i) {
        result *= x;
    }
    return result;
}
```

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

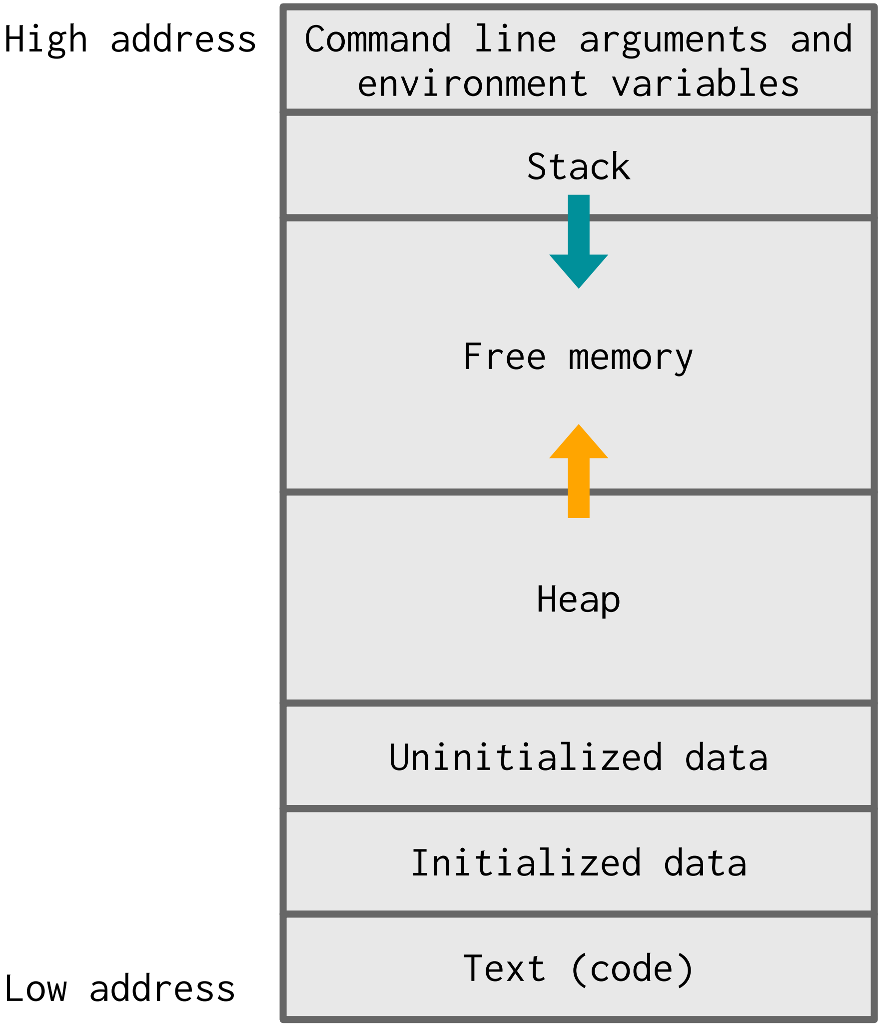
```
int power(int x, int n)
{
    int result = 1;
    do {
        result *= x;
    } while (0 < --n); // you can use the parameter n as a normal variable
    return result;
}
```

MEMORY ALLOCATION, STACK AND HEAP

- Your programs require dynamic memory that your 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:



LIFO stack (Last-In-First-Out), allocated by the OS at program start. Stack frames are created here (e.g. function calls). Easy to manage and much faster than memory allocation on the heap.

Stack and heap grow in opposite direction.

Dynamic memory that grows depending on program. Hard to manage (fragmentation, garbage collection) and involves expensive kernel calls (e.g. malloc or new). Allocating memory on the heap is slow.

Global and static variables. The linker allocates memory in these segments. Those variables can be either initialized or uninitialized.

Executable machine code (instructions), read-only segment.

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:

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

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

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

3. Compile and run:

```
$ g++ stackoverflow.cpp && ./a.out
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:

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

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

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

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`

```cpp
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
    cout << "Type something: "; // send something to stdout
    string received;
    cin >> received; // read from stdin until newline character
    cout << "You typed: " << received << endl;
    return 0;
}
```

```
$ g++ io.cpp && ./a.out
Type something: something
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:

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

void print(ostream &stream, const string message)
{
    stream << message << endl; // append a newline at the end
}

int main(void)
{
    // create an output stream associated to file 'output.txt'.  Note: ofstream
    // is an output file stream object, it inherits from the ostream object.
    // Because of this inheritance we can use the 'ostream &' reference in the
    // print() function above.
    // See https://www.cplusplus.com/reference/fstream/fstream/
    ofstream my_file("output.txt");
    print(my_file, "This message is streamed into my file");
    print(cout, "This message is streamed to stdout");
    print(cerr, "This message is streamed to stderr");
    return 0;
}
```

INPUT/OUTPUT ROUTINES

Example: print function to stream to different objects:

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

See also the following link for format manipulators used together width 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

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