



UNIVERSITY
OF COLOGNE

Software & System Engineering / Software Technology
Abteilung Informatik, Department Mathematik / Informatik

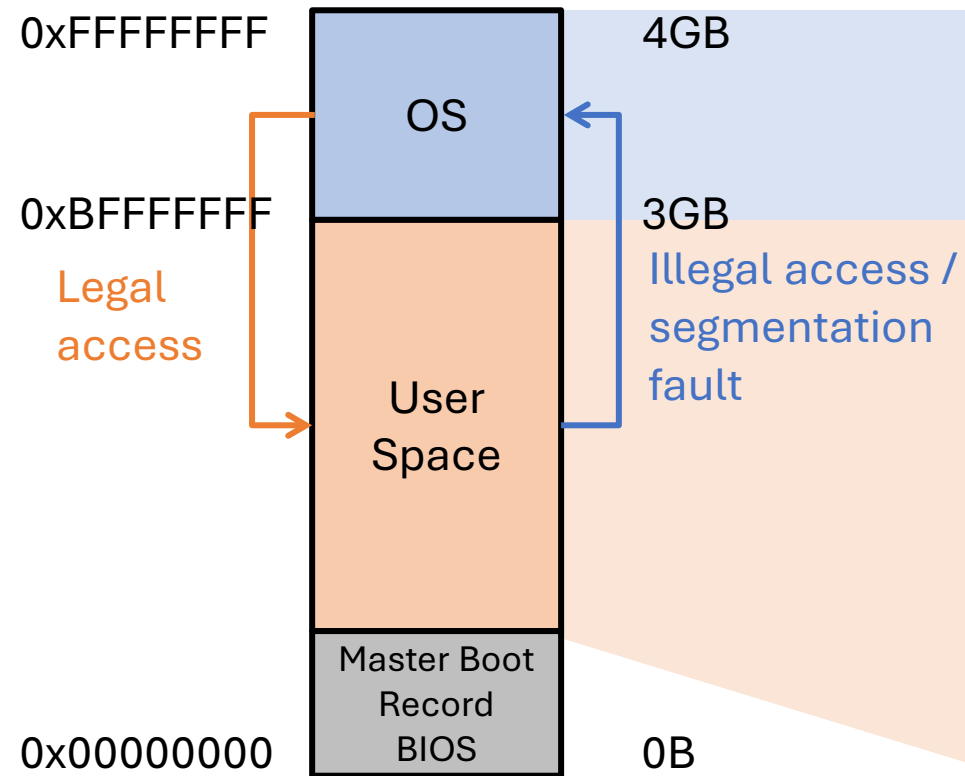
Object-Oriented Software Engineering

Memory Management

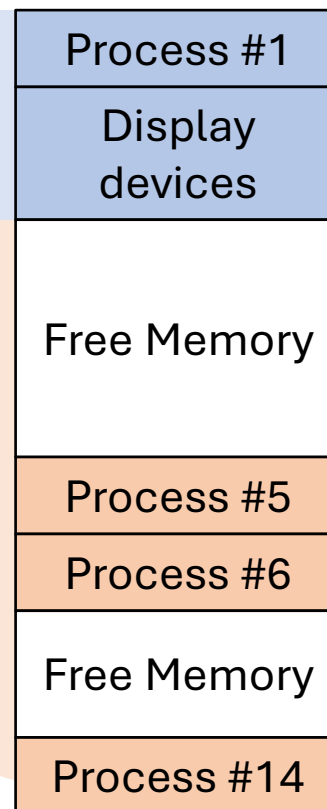
Adrian Bajraktari | bajraktari@cs.uni-koeln.de | SoSe 2024 | 24.04.2024

Memory Layout

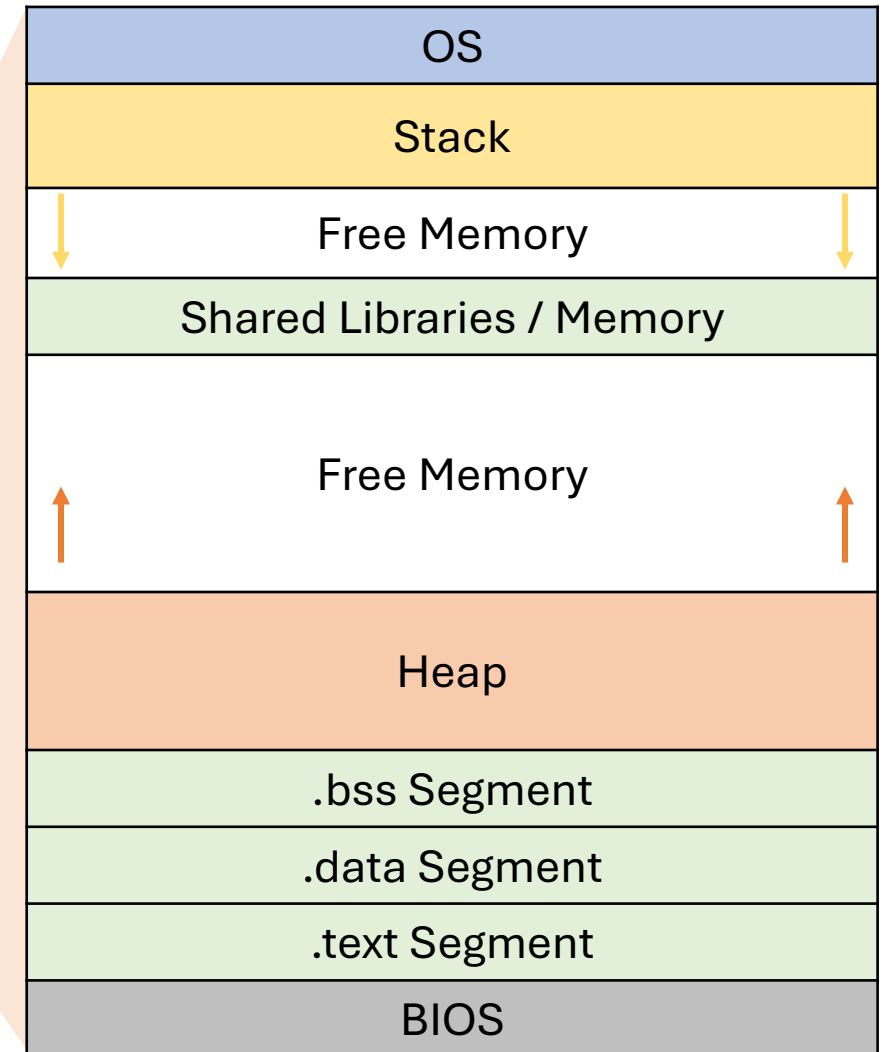
Memory Spaces



Process View



Process Memory



Details of each Memory Section

Part	Allocated by	Contains	Deallocated by	Thread	R/W	Common Errors
Stack	Procedure calls	Parameters, Return Address, non-static local variables	Return or process termination	Specific	R+W	StackOverflow
Heap	malloc() or new	Dynamically created objects	free(), delete or process termination	Shared	R+W	OutOfMemory, Memory Leaks
.bss	Non-initialized static global variables		Process termination		R+W	
.data	Initialized static or global variables		Process termination		R+W	
.text	Procedures, constants	Binary instructions	Process termination		R	

Call by Value Passing

Call by Value

The arguments are evaluated before being passed to the procedure. The value of the argument is copied to the corresponding parameter.

```
int pow(int x) {  
    int result = x * x;  
    x += 5;  
    return result * x;  
}  
  
int x = 5;  
  
println(pow(x)); //250  
println(x);      //5
```

0xA7	5 (x)	5 (x)	5 (x)	5 (x)	5 (x)
0xA6		return address	return address	return address	return address
0xA5		base pointer	base pointer	base pointer	base pointer
0xA4			25 (result = x * x)	25 (result = x * x)	25 (result = x * x)
rdi		5 (x)	5 (x)	10 (x)	10 (x)
rax					250 (result * x)

Stack evolution over time

Call by Reference Passing

Call by Reference

The parameter becomes an alias for the argument. Thus, it is possible to change the original value from within the procedure.

```
int pow(int& x) {  
    int result = x * x;  
    x += 5;  
    return result * x;  
}  
  
int x = 5;  
  
println(pow(x)); //250  
println(x);      //10
```

0xA7	5 (x)	5 (x)	5 (x)	10 (x)	10 (x)
0xA6		return address	return address	return address	return address
0xA5		base pointer	base pointer	base pointer	base pointer
0xA4			25 (result = x * x)	25 (result = x * x)	25 (result = x * x)
rdi		0xA7	0xA7	0xA7	0xA7
rax					250 (result * x)

Stack evolution over time



Pointers

Pointer

Dynamic Memory Management

Dynamic memory management is a necessity in any bigger software project. Dynamic memory is allocated on the heap. Memory on the heap is referenced via pointers on the stack.

Pointer

On language level: A variable that stores the address of block of memory cells, e.g., another variable or an object.

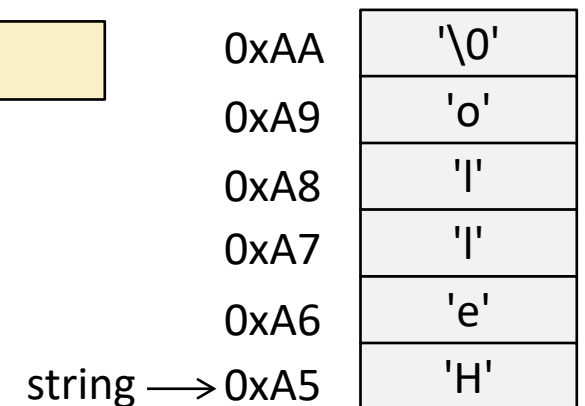
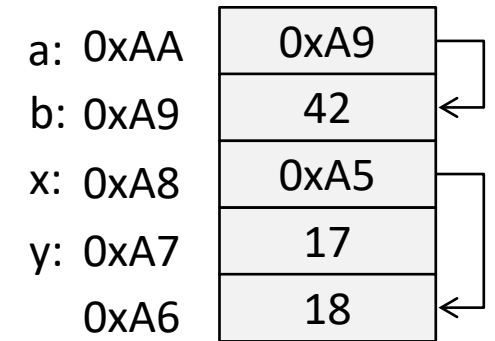
On machine level: A cell, interpreted as an address, that contains the address of another memory cell.

Implementation

In C-like languages that allow usage of pointer, `*p` dereferences a pointer, i.e., we say "get what is at the address stored in `p`" and `&v` gets the address of the variable `v`.

```
int* a;  
int b = 42;  
a = &b; //let a point to b  
  
int* x;  
int y = 17;  
x = (&y - 1); //let x  
//point to address below y  
*x = 18;
```

```
char* string = "Hello";
```



Obtaining Memory

malloc

`malloc` (memory allocation) is a system function that allocates the provided number of bytes of memory cells and returns the address of the first cell. The return type of `malloc` is `void*`, thus we need to cast the correct pointer type accordingly.

free

`free` is a system function that marks the block of memory starting at `ptr` as free so that it can be overridden by other data.

Best Practice

Any memory allocated by `malloc` must be freed by `free` at some point. Attention: Risk of memory leak or data loss.

```
void* malloc(unsigned int length)
void free(void* ptr)
```

```
int* x = (int*) malloc(sizeof(int) * 5);

if(x == NULL) {
    //Error handling
}
else {
    *(x+0) = 1;
    *(x+1) = 2;
    *(x+2) = 3;
    *(x+3) = 4;
    *(x+4) = 5;
    int a = *(x + 1);
    int b = *x;
    free(x);
}
```


Pointers to Procedures

Pointers to Procedures

In C, a "function pointer" is a pointer to the address of a procedure.

```
void proc(int a) {  
    printf("The value: %d\n", a);  
}  
  
int main(int argc, char** argv) {  
    void (*proc_ptr)(int) = &proc;  
  
    (*proc_ptr)(10);  
}
```

```
void proc(int a) {  
    printf("The value: %d\n", a);  
}  
  
int main(int argc, char** argv) {  
    void (*proc_ptr)(int) = proc;  
  
    proc_ptr(10);  
}
```



Structs

Struct

Struct

A struct (or record in other languages) is a unit of variables of arbitrary type. A struct's variables lie next to each other in memory and can be accessed via member access.

Struct Composition

A struct can itself consist of other structs (either by value or by reference).
A struct can not compose itself by value.

Structs in Memory

In C and C-based languages, structs are usually stored in stack memory if not allocated manually on the heap.

Definition:

```
struct Student {  
    char* name;  
    int matNr;  
}
```

Declaration of a struct variable with initialization:

```
struct Student alice = {"Alice", 1688451};
```

Member access:

```
alice.matNr = 42;
```

Pointers to Struct Instances

Pointers to Structs

Most times, struct instances are allocated on the heap using malloc. Member access on struct instances on the heap differs in C from normal member access.

Pointers to Structs

Most times, struct instances are allocated on the heap using malloc. Member access on struct instances on the heap differs in C from normal member access.

By Reference Component Initialization

Allocating memory for a struct does not allocate memory for other struct instances that are components by reference. Each of them must be malloced separately.

Initialization:

```
struct Student* s_heap_pointer =  
    malloc(sizeof(struct Student));
```

```
struct Student* s_heap_pointer =  
    malloc(sizeof(char*) +  
    sizeof(int));
```

Member Access:

```
(*s_heap_pointer).matNr = 42;
```

```
s_heap_pointer->matNr = 42;
```

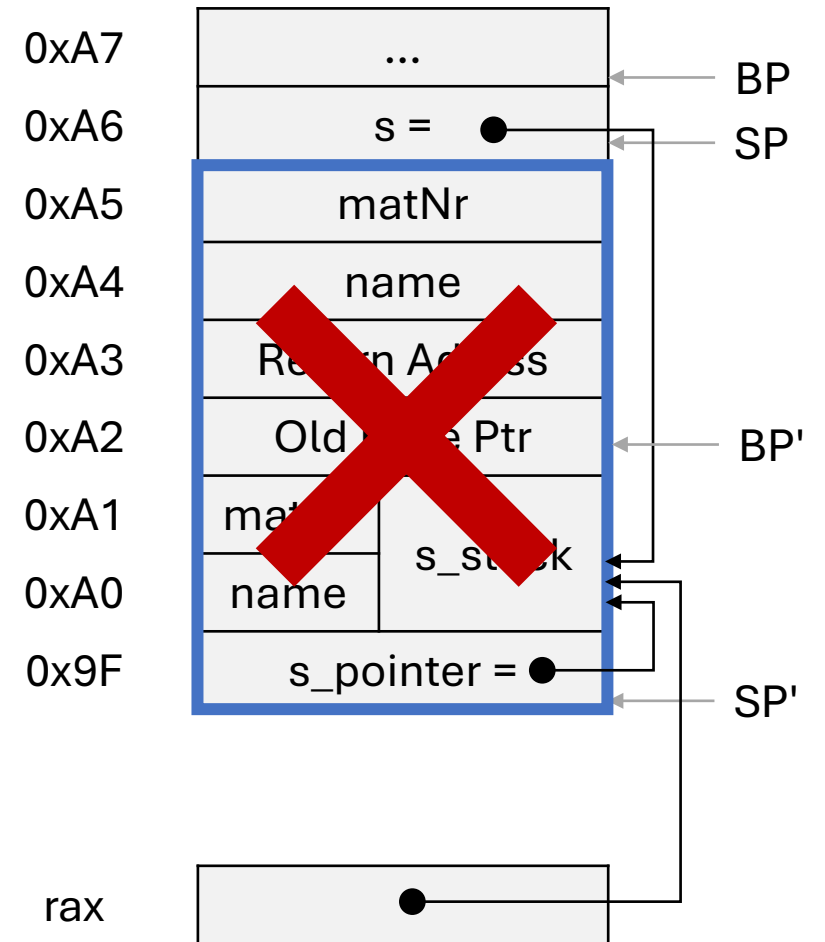
Attention! Structs on Stack

Structs on Stack

Structs instances that are not allocated via malloc are stored in the **stack frame** of the enclosing procedure. Returning pointers to such instances will result in undefined behavior.

```
struct Student* create(char* name, int matNr) {  
    struct Student s_stack = {name, matNr};  
    struct Student* s_pointer = &s_stack;  
  
    return s_pointer;  
}
```

```
int main() {  
    struct Student* s = create("Alice", 8329583);  
  
    printf("%s, %d", s->name, s->matNr);  
    return 0;  
}
```





Destructors and Garbage Collection

Destructors

Destructor

In languages with manual memory management, like C++, a destructor is a class method that is called when an object is no longer needed.

The destructor defines logic that will be executed before the objects destruction, like closing connections or returning system resources.

Implementation

In most cases, no special care is needed for an object's destruction, so the constructor is empty and thus omitted (the compiler adds a default constructor).

Theoretically, it is the developers responsibility to delete an object when it is no longer needed. Practically, modern compilers can insert object deletion where feasible. Best practice: Always explicitly delete objects.

```
class Student {  
private:  
    ...  
public:  
    ...  
    ~Student() {  
  
    }  
}
```

```
Student* student = new...  
...  
delete student;
```


Garbage Collection

```
{  
    Time t;  
    t = new Time();  
}
```

Stack

something
Something else

Heap

Garbage Collection

```
{  
    Time t;  
    t = new Time();  
}
```

Stack

something
Something else
t = null

Heap

Garbage Collection

```
{  
    Time t;  
    t = new Time();  
}
```

Stack

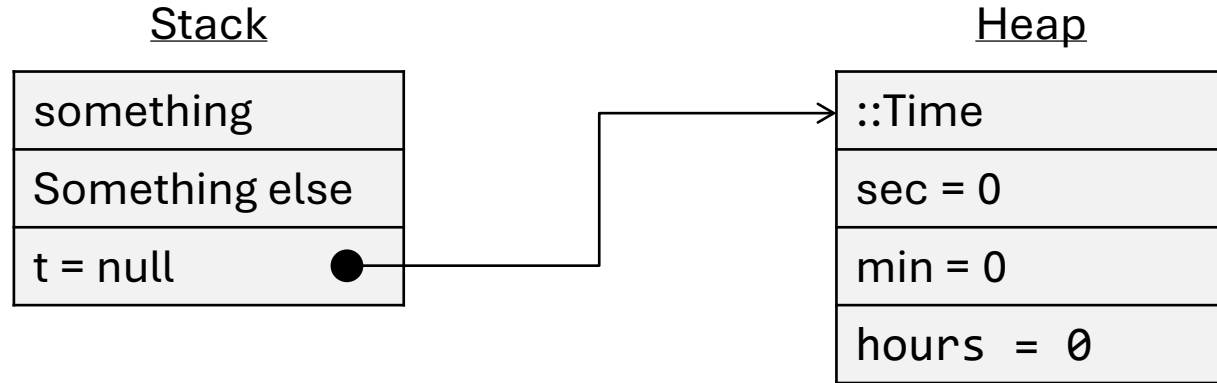
something
Something else
t = null

Heap

::Time
sec = 0
min = 0
hours = 0

Garbage Collection

```
{  
    Time t;  
    t = new Time();  
}
```




Garbage Collection

```
{  
    Time t;  
    t = new Time();  
}
```

Stack

something
Something else
t = null

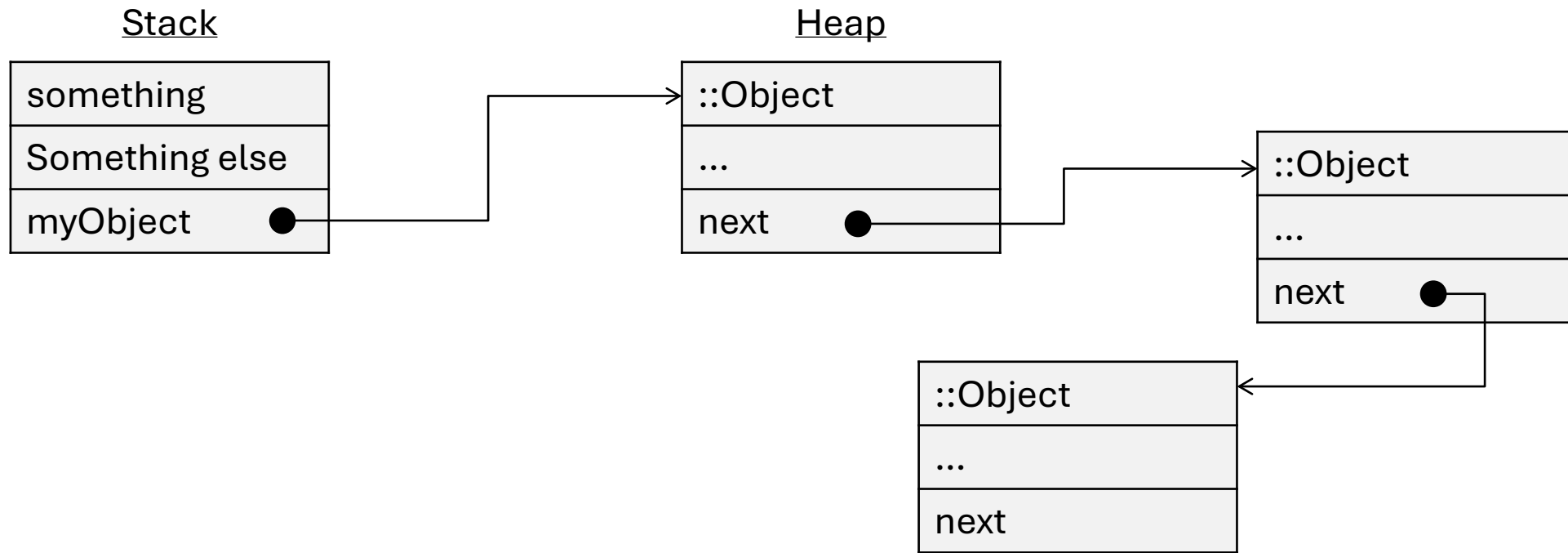
Heap

::Time	
sec = 0	
min = 0	
hours = 0	

Reachability

Garbage Collection

Objects on the heap live as long as they are **reachable**, either directly or transitively from attributes of other objects. When they are not reachable anymore (**dead**, **garbage**), they are removed (**collected**) by the **garbage collection**.



Garbage Collection

Garbage Collection
<ul style="list-style-type: none">• The concrete implementation of the garbage collection algorithm is not prescribed. There are many variants.• A simple approach: Mark and Sweep. The runtime system marks all objects (transitively) reachable from the stack as alive. Then, it sweeps, i.e., deletes all remaining dead objects and removes the markings.• An additional step may also move all alive objects to defragment memory.• Another additional step may move objects that are often used together to neighboring memory cells.

Benefits	Drawbacks
<ul style="list-style-type: none">+ Developer does not need to care whether objects are alive.+ No risk of memory leak.+ Efficient implementations.	<ul style="list-style-type: none">- Risk of keeping objects alive unintentionally.- Not efficient enough for real time applications.

Coding Time!

OOP in C?

Observation

- A module encapsulates all data and procedures that operate on that data.
- A struct defines all variables of a unit. There can be arbitrary instances of that unit with different values.
- In C, if a struct B is a structurally subset of another struct A, B can substitute A via casting.
- A struct can store procedure pointers. These can be replaced with arbitrary procedures that fulfill the required interface.

Coding Time!

Comparison Java vs. C endless object creation

```
struct T {
    char* name;
    int age;
    int matNr;
};

int main() {
    while(true) {
        struct T* t = (struct T*) malloc(sizeof(char*) + sizeof(int) + sizeof(int));
        t->name = "Testus";
        t->age = 42;
        t->matNr = 42424242;
    }
    return 0;
}
```

The following chapters deal with topics that should be known but mostly focus on the memory management aspects.



Arrays

Arrays in Java

Arrays

In Java, arrays of type T are instances of type T -Array. T can be any primitive or non-primitive type.

When declaring an array, the subscript operator `[]` is either

- directly after the type (Convention)
- or after the identifier ("C style")

The index is an expression that evaluates to a positive `int`. An array's maximum size is $2^{31}-1$ in Java. The length of the array is stored in `array.length`. Accessing indices < 0 or \geq length produces `ArrayIndexOutOfBoundsException`.

```
int[] integers = new int[]  
    {4, 3, 42, 1, 3, 91, 4, 15, 21, 70};
```

4	3	42	1	3	91	4	15	21	70
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

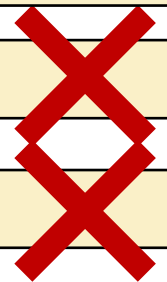
Array initializer:

```
int[] integers =  
    new int[] {42, 13};
```

```
int[] integers = {42, 13};
```

```
integers = {42, 13};
```

```
method({42, 13});
```

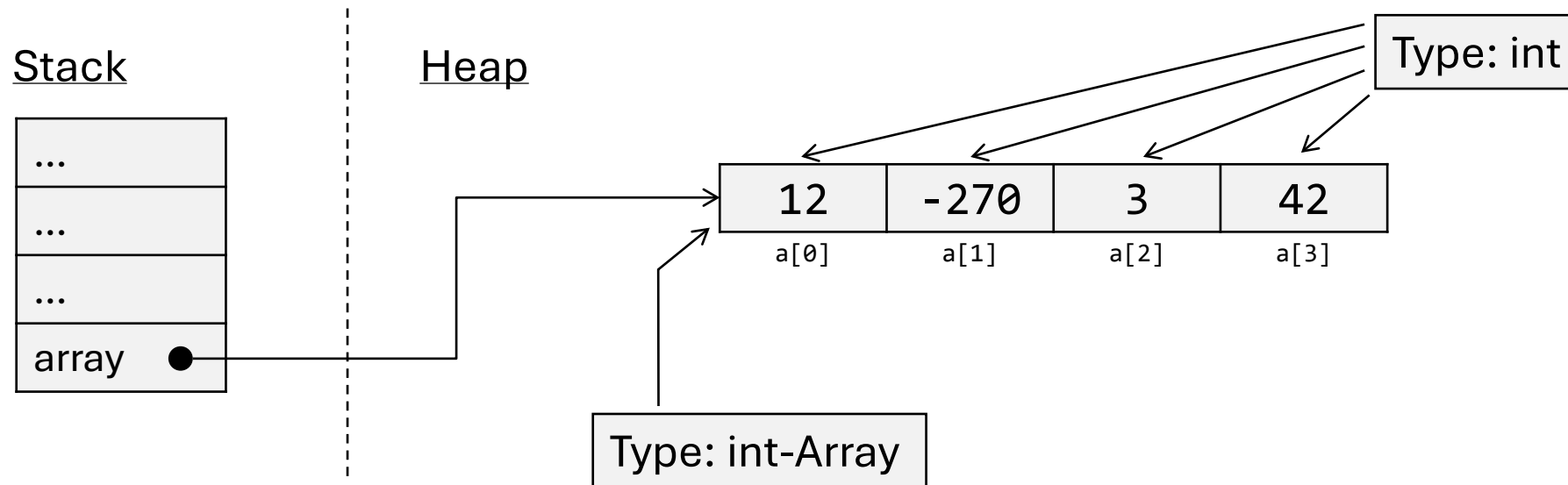


Via Constructor and explicit size:

```
int[] integers = new int[2];
```

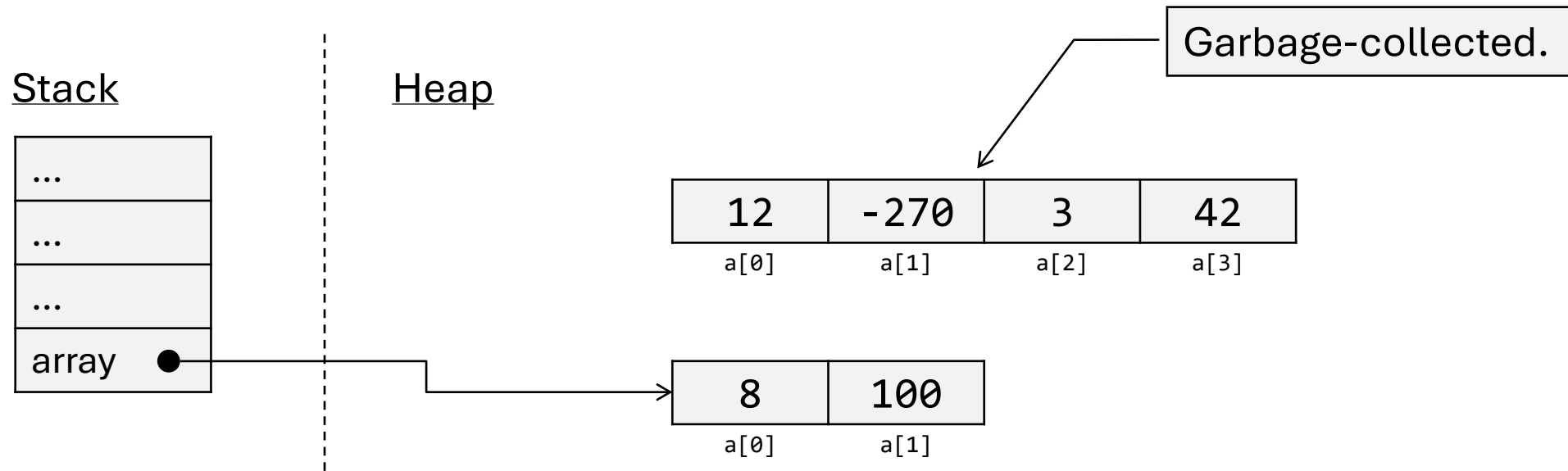

Memory Layout

```
int[] array = { 12, -270, 3, 42 };
```



Memory Layout

```
int[] array = { 12, -270, 3, 42 };  
array = new int[] { 8, 100 };
```



"Multi-Dimensional Arrays" and Jagged Arrays

```
float[][] matrix = new float[n][m];
```

```
Block[][][] chunk = new Block[n][m][k];
```

```
chunk;           //"cube"  
chunk.length;    //-> n  
chunk[0];        //"plane"  
chunk[0].length; //-> m  
chunk[0][0];     //"line"  
chunk[0][0].length; //-> k  
chunk[0][0][0];  //single value
```

```
int[] array = new int[4][2] {  
    {1, 2},  
    {3, 4},  
    null,  
    {7, 8}  
};
```

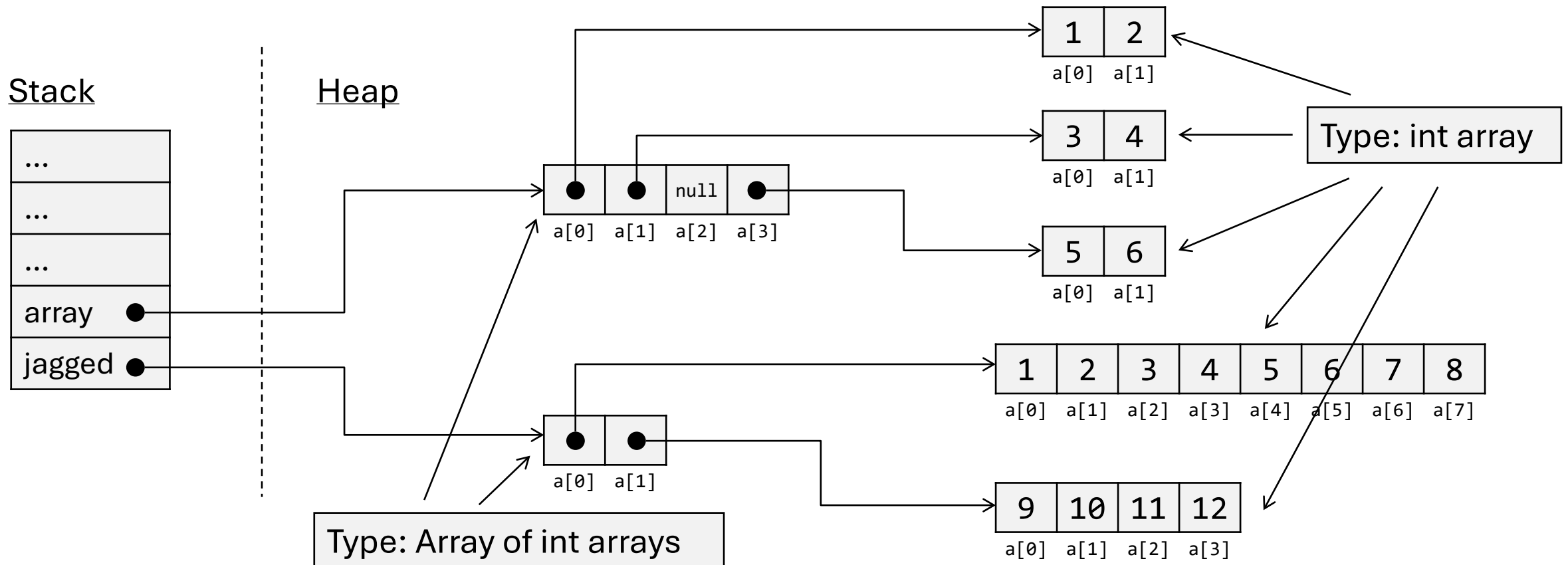
```
float[][] jagged = new float[2][]; //size 2  
jagged[0] = new float[8];          //size 8  
jagged[1] = new float[4];          //size 4
```

```
float[] array = new float[][] {  
    {1, 2, 3, 4, 5, 6, 7, 8},  
    {9, 10, 11, 12}  
};
```

Memory Layout

```
int[] array = new int[4][2] { {1, 2}, {3, 4}, null, {7, 8} };
```

```
float[] array = new float[][] { {1, 2, 3, 4, 5, 6, 7, 8}, {9, 10, 11, 12} };
```



The Class Arrays

Provides some class methods to handle arrays.

<code>int compare(T[] arr1, T[] arr2)</code>	<code>//compare two arrays lexically</code>
<code>T[] copyOf(T[] arr, int newlength)</code>	<code>//create new array as copy of arr</code>
<code>void fill(T[] arr, int val)</code>	<code>//fill array with val</code>
<code>String toString(T[] arr)</code>	<code>//create shallow String representation of //array</code>
<code>String deepToString(T[] arr)</code>	<code>//create a deep String representation //(recursive)</code>
<code>boolean equals(T[] arr1, T[] arr2)</code>	<code>//check whether the arrays are shallow //value-equal</code>
<code>boolean deepEquals(T[] arr1, T[] arr2)</code>	<code>//check whether the arrays are deep //(recursive) value equal</code>

Enhanced for Loop and Varargs

Enhanced for

Java provides an **enhanced for loop** for read-only iterating over collections.

```
for(Student s: students){  
    print(s);  
}
```

VarArgs

We can define arbitrary many parameters of the same type using **varargs** (ellipse notation) in last position once per method.

```
public void method(int a, float b, boolean... bArray) {  
    if(b[b.length - 1]) {  
        ...  
    }  
}
```

```
methode(5, 7.3f, true, true, false);  
methode(5, 7.3f, true);  
methode(5, 7.3f, true, false, false, false, false);  
methode(5, 7.3f, false, true, true, false);
```



Strings

The Class String

String

`String` is not primitive type in Java. String literals are instances of `String`. Assigning literals is equivalent to calling the constructor.

String instances store their length and are thus not terminated with `'\0'` as in C.

String instances are **immutable**. For **mutable** strings use `StringBuilder` or `StringBuffer`.

Comparing String

Comparing two strings via `==` is generally discouraged.

The JVM pre-generates String instances for constant String literals and concatenated constant String literals in a **constant pool**. The same literals thus always reuse the same instance.

```
String string = "abc";  
String string = new String("abc");
```

```
String string1 = "abc";  
String string2 = "abc";  
print(string1 == string2);
```

true

```
print("a" + "b" == string2);
```

true

```
String string1 = "hallo";  
String string2 = "ha";  
String string3 = "llo";  
  
print(string1 == (string2 + string3));
```

false

```
print(string1.equals(string2 + string3));
```

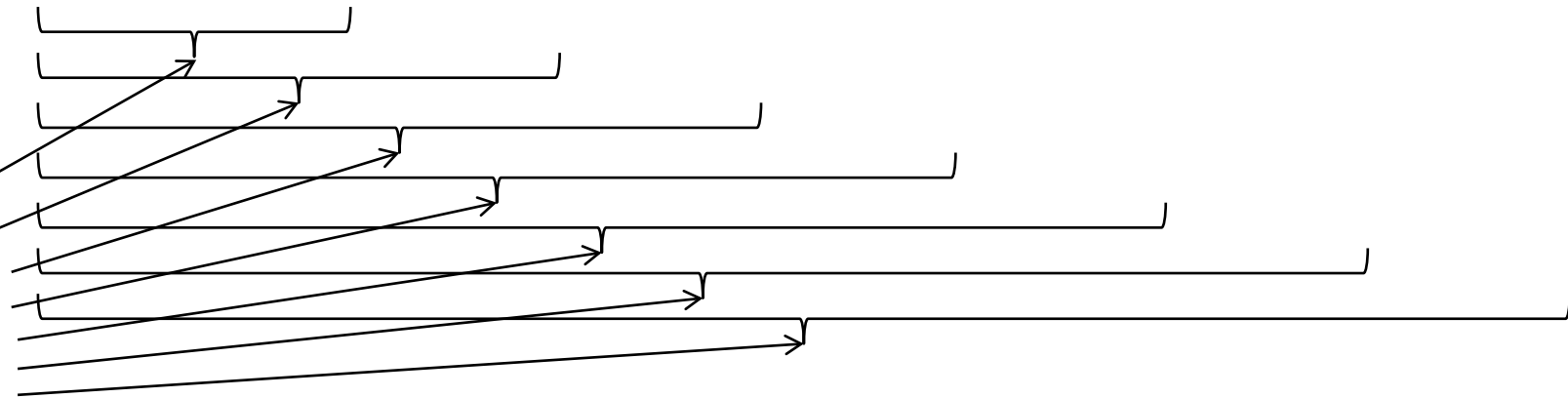
true

Concatenation

Using + concatenates two strings. Using many concatenations is inefficient. Instead, prefer `StringBuilder` or `StringBuffer`.

```
String[] s = {"abc", "def", "ghi", "jkl", "mno", "pqr", "stu", "vwx"};  
String s_t = s[0] + s[1] + s[2] + s[3] + s[4] + s[5] + s[6] + s[7];
```

New instance of
String each time.



toString()

All classes inherit `toString()` from `Object`. Per default, it only prints the object's `hashCode`.

We override it with a more sensible implementation. IDEs provide automatic generation of `toString()` with some standard templates.

```
public class Student { ...  
    public String toString() {  
        return this.name + ", "  
            + this.matNr  
            + " is in semester "  
            + this.semester  
            + " and currently listens to "  
            + this.currentLecture;  
    }  
}
```

Compiler adds
`toString()` call.

`System.out.println(student);`

StringBuilder vs. StringBuffer

Both realize the same methods, but

- When using threads and the string creator should be shared between threads, use `StringBuffer` (slower implementation, thread-safe).
- Else, use `StringBuilder` (faster implementation, not thread-safe).

```
StringBuilder()           //Create instance with initial space of 16
StringBuilder(int length) //Create instance with initial space of length
StringBuilder(String str) //Create instance with copy of str and initial
                          //space for 16 more
```

```
StringBuilder append(String s)
```

Also defined for primitive types. Returns itself.

```
StringBuilder sb = new StringBuilder();
sb.append("Hello ")
  .append("OOSE ")
  .append("Lecture!");
```

Instance Methods of Class String

<code>int length()</code>	<code>//length of string</code>
<code>boolean isEmpty()</code>	<code>//no chars?</code>
<code>boolean isBlank()</code>	<code>//only whitespace?</code>
<code>char charAt(int pos)</code>	<code>//get char at pos</code>
<code>boolean contains(CharSequence seq)</code>	<code>//contains the sequence?</code>
<code>int indexOf(char c)</code>	<code>//index of first occurrence of char</code>
<code>int indexOf(String s)</code>	<code>//index of start of first occurrence of sequence</code>
<code>int lastIndexOf(char c)</code>	<code>//index of last occurrence of char</code>
<code>int lastIndexOf(String s)</code>	<code>//index of start of last occurrence of sequence</code>
<code>char[] toCharArray()</code>	<code>//transform to char array</code>
<code>String concat(String s)</code>	<code>//concatenate strings as new string</code>
<code>String toLowerCase()</code>	<code>//transform to lower case only</code>
<code>String toUpperCase()</code>	<code>//transform to upper case only</code>
<code>String trim()</code>	<code>//remove spaces</code>
<code>String strip()</code>	<code>//remove whitespaces</code>
<code>String stripLeading()</code>	<code>//remove leading whitespaces</code>
<code>String stripTrailing()</code>	<code>//remove trailing whitespaces</code>
<code>String subSequence(int start, int end)</code>	<code>//get substring from start index to end index</code>
<code>String replace(char a, char b)</code>	<code>//replace all occurrences of char a with b</code>
<code>String replace(CharSequence a, CharSequence b)</code>	<code>//replace all occurrences of sequence a with b</code>
<code>boolean equals(String s)</code>	<code>//are strings value-equal?</code>
<code>boolean equalsIgnoreCase(String s)</code>	<code>//are strings value-equal not case-sensitive?</code>
<code>int compare(String s)</code>	<code>//compare strings lexically</code>
<code>String replaceAll(String regex, String repl)</code>	<code>//replace all occurrences matching regex with repl</code>



Wrapper Classes and System

Wrapping Primitive Types

Java provides **wrapper class** to treat primitive types like reference types.

Auto-boxing: `Integer integer = 42;`

Why wrapper classes?

- Provide methods and constants for primitive types.
- Less overloading variants of a method needed.
- Synchronization in Multithreading.
- Generics.

Interesting Methods and Constants of Wrapper Classes

```
Integer.parseInt(String s)           //same for Byte, Short, Long
Integer.MAX_VALUE                     //same for Byte, Short, Long
Integer.MIN_VALUE                     //same for Byte, Short, Long
Float.parseFloat(String s)           //same for Double
Float.MAX_VALUE                       //same for Double
Float.MIN_VALUE                       //smallest float/double value
Float.MIN_NORMAL                      //smallest float/double value > 0
Float.NaN                             //Not a number, same for Double
Float.NEGATIVE_INFINITY               //-inf, same for Double
Float.POSITIVE_INFINITY               //+inf, same for Double
Boolean.parseBoolean(String s)
Character.MAX_VALUE
Character.MIN_VALUE
Character.isAlphabetic(char c)
Character.isDigit(char c)
Character.isLetter(char c)
Character.isWhiteSpace(char c)
Character.isUpperCase(char c)
Character.isLowerCase(char c)
Character.toUpperCase(char c)
Character.toLowerCase(char c)
```

} Can be used to
check whether
float/double
variable has
sensible value.

System

The class `System` provides two methods to measure time:

```
long System.currentTimeMillis()    //system time since the epoche (01.01.1970,  
                                   //00:00:00) in milliseconds  
long System.nanoTime()            //system time since unspecified point in time  
                                   //origin
```

`System` stores standard print streams:

- `System.out`: Standard output, console
- `System.err`: Standard error output, console
- `System.in`: Standard input, console



Iteration vs. Recursion

Iteration vs. Recursion

Iteration (Loops)

The arguments are evaluated before being passed to the procedure. The value of the argument is copied to the corresponding parameter.

Recursion

The parameter becomes an alias for the argument. Thus, it is possible to change the original value from within the procedure.

Design Decision

- Recursion is good for multithreading.
- Recursion is slower and uses increasingly more stack memory. Danger of StackOverflow.
- In functional languages, tail recursion is optimized to turn into loops in compiled code.

```
while(condition) {  
    //code  
}
```

```
returnType name(type1 para1) {  
    //code  
    name(x);  
    //code (if not tail recursion)  
}
```

Iteration vs. Recursion: The Stack Memory

Iteration overrides the same cells on the stack its body is using in each iteration.

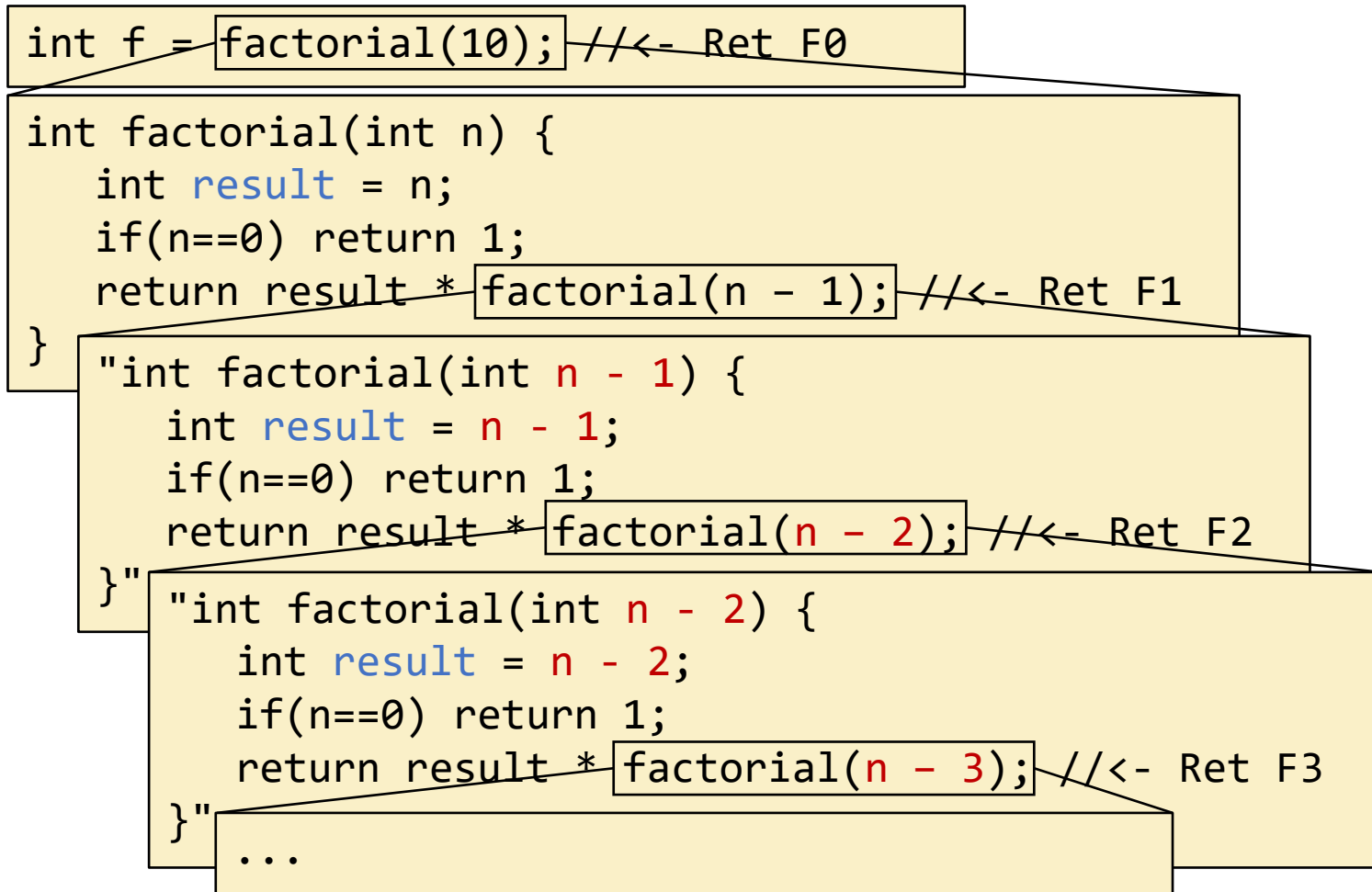
```
int f = factorial(10); //<- Ret F0
```

```
void factorial(int n) {  
    int result = n;  
    for(int i = 1; i < n; i++)  
        result *= i;  
    return result;  
}
```

		It. 1	It. 2	It. 3	It. 4
0xA2	Ret F0				
0xA1	result	= n	= n*1	= n*1*2	= n*1*2*3
0xA0	i	= 1	= 2	= 3	= 4

Iteration vs. Recursion: The Stack Memory

Recursion always creates a new stack frame.



0xA7	Ret F0	Stackframe "n"
0xA6	result (n)	
0xA5	Ret F1	Stackframe "n-1"
0xA4	result (n - 1)	
0xA3	Ret F2	Stackframe "n-2"
0xA2	result (n - 2)	
0xA1	Ret F3	
0xA0	...	

Side Effects of Operations

Pure Function

A pure function is a subroutine that only returns its output and whose output only depends on the input.

Side Effect

A side effect is any activity or change a procedure causes besides returning values that can be observed from the outside.

Examples of Side Effect

- Reading or writing variables outside the procedure scope, e.g. state of an object.
- Reading from / writing to the console.
- Randomness.

$$f(3) + g(3) \neq g(3) + f(3)$$

because then:

$$f(3) = 4 + 3 = 7,$$

$$g(3) = 4 + 3 = 7, \quad 7 + 7 = 14$$

$$g(3) = 7 + 3 = 10,$$

$$f(3) = 4 + 3 = 7, \quad 10 + 7 = 17.$$

```
int y = 7;

int f(int x) {
    y = 4;
    return y + x;
}
```

```
int g(int x) {
    return y + x;
}
```

```
int a = 3, b = 7;
int i = ++a; //i = 4
int j = b++; //j = 7
```