50006 - Compilers - (Dr Dulay) Lecture 5

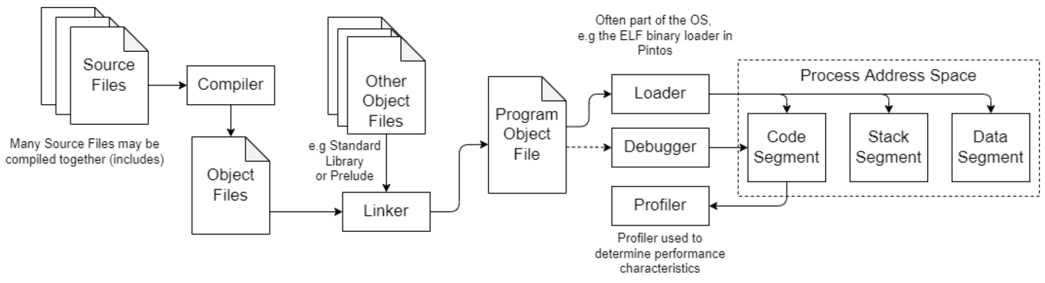
Oliver Killane

14/04/22

Lecture Recording

Lecture recording is available here

Classic Approach to Compilers


Source Files
Many Source Files may be compiled together (includes)
Compiler
Object Files
Other Object Files
e.g Standard Library or Prelude
Linker
Program Object File
Often part of the OS, e.g the ELF binary loader in Pintos
Loader
Debugger
Profiler
Profiler used to determine performance characteristics
Process Address Space
Code Segment
Stack Segment
Data Segment

Data Representation

Definition: Primitive Types

The most basic types, supported by the architecture the compiler targets.

Type	**Bytes**	**Representation**
Boolean	1	0 for false and 1 for true.
Integer	1,2,4,8	Typically 2s-complement.
Unsigned Integer	1,2,4,8	Basic unsigned binary.
Float/Real	4,8,16	IEEE Floating Point.
Char	1,2	For ascii (byte) and unicode.

- Compilers may align types for more optimal memory access, sometimes this is enforced by the target architecture.
- Some languages support pointer types (e.g C), however this makes type checking difficult (consider pointer arithmetic). As a result more modern language use type-checkable object references.

Definition: Alignment

Typically some address is considered n-byte aligned when it is a multiple of n bytes.

- Often data is aligned by the size of data the processor can load in a single instruction. This is done to ensure loading some data requires the minimal number of memory accesses.
- Some architectures enforce alignment for this reason, particularly for the stack.
- When aligned to a multiple of two 2^k, the first k bits of the address will be zero. For example alignment of pages in memory allow for page table entries to use the bits that will always be zero (due to alignment) to be used for other information (e.g reference, supervisor, read/write)

Definition: Structs/Records

Data types consisting of groups of fields/members of other types.

- Fileds typically allocated in a contiguous block of memory.
- Alignment often used to space fields.
- Some languages order fields by their position in code (e.g C) while others can reorder and optimise.
- Tuples are effectively anonymous structs.

If no padding/alignment is used, and the struct/record is kept in order as expressed by the source code (e.g typical of C):

$$\begin{aligned} Record &= (Field_1, Field_2, \dots, Field_n) \\ Size(Record) &= Size(Field_1) + Size(Field_2) + \cdots + Size(Field_n) \\ Address(Field_k) &= StartAddress(Record) + Size(Field_1) + \cdots + Size(Field_{k-1}) \end{aligned}$$

Struct Representation

As previously mentioned some languages reorder and optimise structs/records.

Rust not only does this, but allows the programmer to specify different representation schemes through macros. (See Rust alternative representations.)

Definition: Arrays

A contiguous section of memory populated by n variables (elements) of the same type.

- Can have elements aligned
- Some languages associated arrays with auxiliary data (e.g length for bounds checking).

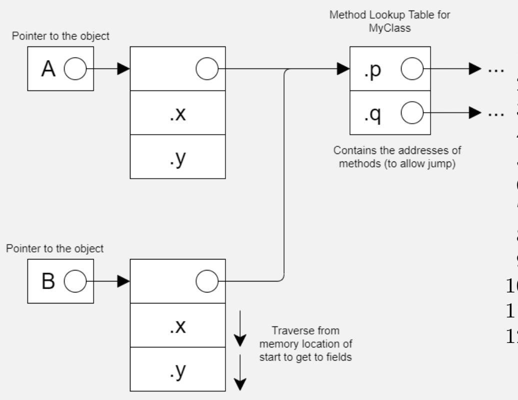
A basic array implementation (elements not aligned/padded):

$$\begin{aligned} Array[Type] &= Element[Type]_1, \dots, Element[Type]_n \text{ (Indexed as 0 to } n-1\text{)} \\ Size(Array) &= Size(Type) \times n \\ Address(Element_k) &= StartAddress(Array) + k \times Size(Type) \end{aligned}$$

Definition: Objects

Implemented as a reference to a record, but with a pointer to a **method lookup table**(**MLT**) for that class (needs to consider inheritance).

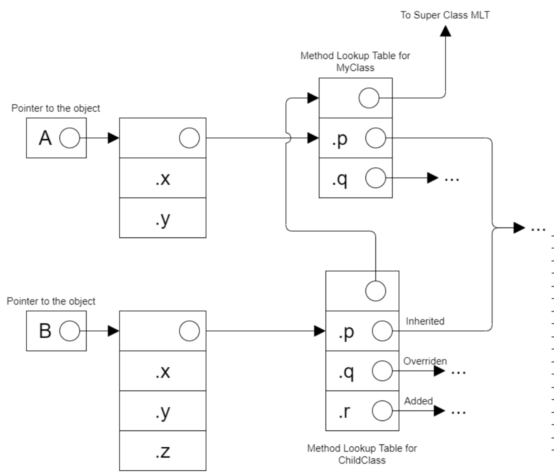
- When calling a method of an object, traverse to the object's **MLT**. Then jump to the method, having placed the first argument as a pointer to the object.
- Access fields just like a struct.
- The indirection required for accessing values, calling methods adds overhead.


Pointer to the object
A
.x
.y
Method Lookup Table for MyClass
.p
.q
...
...
Contains the addresses of methods (to allow jump)
Pointer to the object
B
.x
.y
Traverse from memory location of start to get to fields

```
class MyClass {
    int x;
    int y;

    method ... p(...) {...}
    method ... q(...) {...}
}

MyClass A = new MyClass();
MyClass B = new MyClass();

B.q(...) // MyClass.q(&B, ...)
```

Class Inheritance

For single inheritance we include the parent's fields and methods in the struct and **MLT**.



```
class MyClass {
    int x;
    int y;

    method ... p(...) {...}
    method ... q(...) {...}
}

class ChildClass {
    int z;

    override method ... q(...) {...}

    method ... r(...) {...}
}

MyClass A = new MyClass();
MyClass B = new ChildClass();
```

- Pointers in the **MLT** go to the method (potentially a method also pointed to by the parent if no overriden)
- New added methods are in the **MLT**
- We chain the child/subclass's **MLT** to the parent/superclass' so that we can check for types (e.g is this an *instanceof* another class, a child etc) at runtime using the **MLT**. We could also have used type descriptors or other implementations.
- We preserve the layout of the parent/super class in the child (only appending methods and fields).

Dynamic Binding

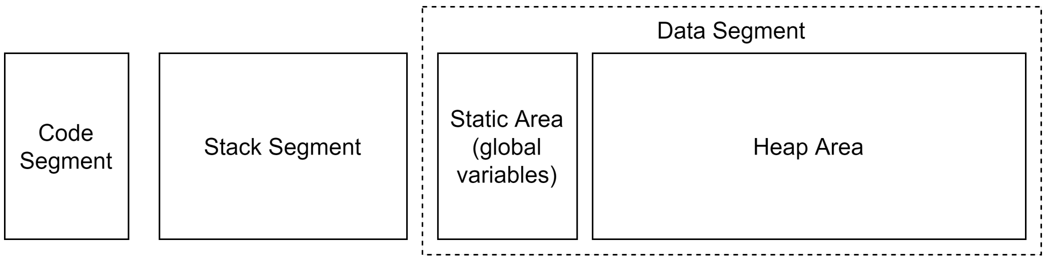
As we only append fields and entries to the **MLT** when inheriting, it is possible to set a variable of type parent class to a value of type child class.

In the example below, we can see that a new field is added in B, (the inherited $A.n$ is still available). The **MLT** of the B class contains the overriden *get* function.

```
public class OOPEXample {
    public static void main(String args[]) {
        A a = new B();
        // As get returns the class's n, and the get method is B's we get 2.
        // However a.n refers to A.n, which is 1. The field n is inherited from
        // A, B adds another field called n.

        // Outputs: "a = new B(); a.get() -> 2, a.n -> 1"
        System.out.printf("A a = new B(); a.get() -> %d, a.n -> %d", a.get(), a.n);
    }
}

/**
 * A:
 * -> MLT: A.get  (defined)
 * - int A.n      (defined)
 */
```

```java
class A {
    public int n = 1;
    public int get() { return n; }
}

/**
 * A:
 * -> MLT: B.get  (override)
 * - int A.n      (inherited)
 * - int B.n      (added)
 */
class B extends A {
    public int n = 2;
    public int get() { return n; }
}
```

Program Address Space



Typically programs divide their address space into several segments (each potentially multiple pages large).

- By placing static data, code and stack space in different pages (in different segments/parts of the address space) they can have different permissions (e.g code is read-execute only, stack cannot be executed, etc)
- Global variables can be placed in a fixed location in the static area.
- Constants, **MLT**s and other global data can be kept in the static area.
- Local variables are stored on the stack in the stack frame of the function they are local to. They are allocated for the duration of the call.
- The stack can also contain arguments sent to the function, as well as information such as the address of the object (for methods) and slots for temporarily storing registers.
- Some architectures such as IA32 have a special register for the start address of the current stack frame (in IA32 it is the EBP register).

Heap Management

Lecture Recording

Lecture recording is available here

We can heap allocate variables (heap variables, also called dynamic variables). Here they can remain for any period of time, and be referenced from anywhere within the program (unlike local variables which are bound to a given function).

Heap variables must be allocated (e.g by *new* in java, or *malloc* in C/C++)

Programming languages manage memory in 4 different ways:

Explicit	C, ASM	Programmer determines when memory is freed, and how it is accessed. This is difficult, but allows for very fast code as the programmer can highly optimize their memory usage.
Garbage Collection	Java, Haskell, Go	The compiler adds code to check what objects are no longer in use, and to free them. This is done at runtime and has considerable overhead.

Rust Superiority

Rust's main memory management tool is Lifetimes (though it does use smart pointers with reference counting, and allow for explicit memory management also).

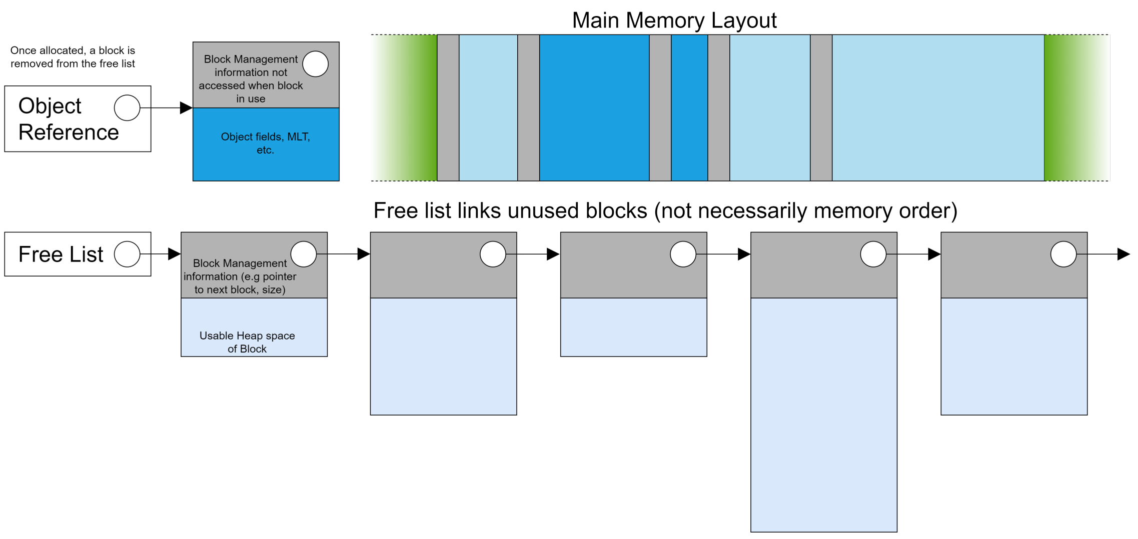
All references are qualified with information on mutability, and where the reference will be valid. The compiler uses these to ensure memory safety and insert required frees at compile time.

This makes the compiler more complex, but is safer than a GC (fewer bugs, will refuse to compile unsafe code (e.g with data races)), with performance almost identical to explicit memory management.

Explicit Heap Allocation

We maintain a structure containing the free blocks and information associated with each block. Here we will consider a simple linked list.

- Maintaining several lists for different sizes can reduce the search time.
- Should allocate blocks just large enough.
- For extensible arrays (vectors, mutable strings) we should allocate with the expectation that more memory will be used (allocate more than currently required).
- Some architectures require alignment of values, this must be considered. Requirements aside, as previously mentioned not-aligning can reduce performance.
- Heap allocation is worse for locality (e.g than the stack) and hence can potentially have worse cache locality, affecting performance.


Main Memory Layout
Once allocated, a block is removed from the free list
Object Reference
Block Management information not accessed when block in use
Object fields, MLT, etc.
Free list links unused blocks (not necessarily memory order)
Free List
Block Management information (e.g pointer to next block, size)
Usable Heap space of Block

```
def allocate(size: int) -> Address:
    # Search through the blocks, can use a better algorithm:
    block = free_blocks.search(size)
    if block.size == size:
        # found exactly the size needed
        free_blocks.remove(block)
        return block.start_address
    elif block.size > size:
        # split the block
        (size_block, other) = free_blocks.split(size)
        free_blocks.remove(size_block)
        return block.start_address
    else:
        # we need more memory
        if os.give_me_more_mem():
           # try again with allocation
        else:
            # OS could not provide
            return NULL

# Can be implemented to takes the block size as a parameter also
def deallocate(ptr: Address):
    # Check free is valid
    if free_blocks.check(address):
        # return block to free blocks, coalesce/merge with another block to
        # reduce fragmentation
        free_blocks.insert_and_coalesce(ptr)
```

Garbage Collection

Definition: Garbage Collector

A part of the language's runtime that ensures heap allocated variables are properly de-allocated when they are no longer used. Many different **GC** algorithms are employed by many different languages.

GCs have several requirements:

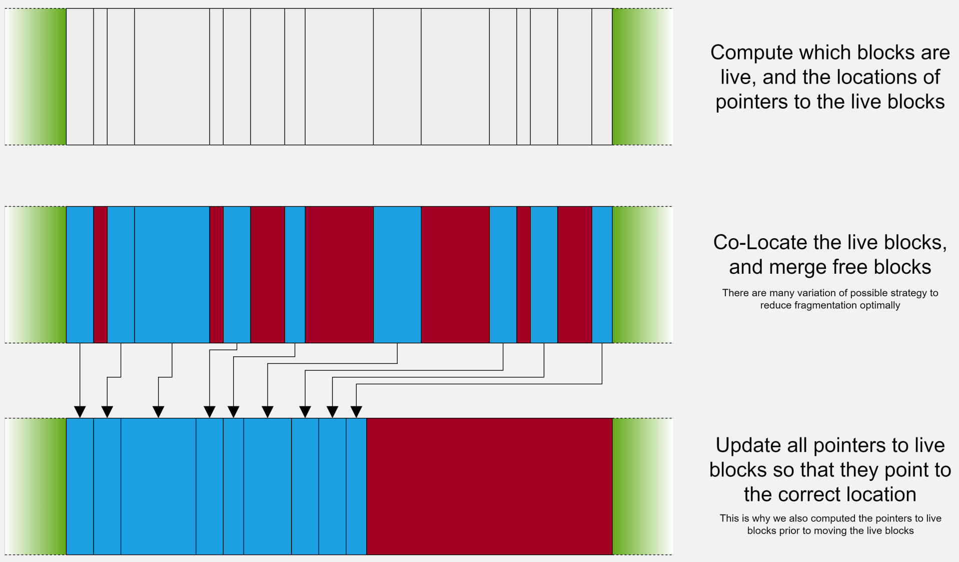
- **Correctness** The **GC** must never collect (de-allocate) live data (in use).
- **Performance** Must be fast & have low memory overhead to reduce the impact on program performance.
- **Compiler Supported** In order to function the compiler needs to provide the garbage collector information for:
  - Which variables point to the heap.
  - The pointers contained in each block (e.g an object contains a reference to an object).

  The compiler can either provide this data directly (type description data for objects), or generate special subroutines for the **GC** to call.

Garbage collectors can be run as **1-shot** (pause the program, run the **GC** and then resume), **on-the-fly** (as the program is run) or concurrent (runs on a different thread).

Definition: Heap Compaction

After **GC** de-allocates many blocks, the heap may be fragmented (lots of small free blocks). Hence much memory may not be free, but it may not be possible to allocate for an object as there is no block large enough.


Compute which blocks are live, and the locations of pointers to the live blocks
Co-Locate the live blocks, and merge free blocks
There are many variation of possible strategy to reduce fragmentation optimally
Update all pointers to live blocks so that they point to the correct location
This is why we also computed the pointers to live blocks prior to moving the live blocks

Definition: Reference-Counting Garbage Collector

The block's management/housekeeping information contains a **reference count**. When a reference is made to an object, the reference count of the block containing it is increased, when that reference is removed, this is decreased.

When the reference count is decremented to zero, we know to free the block.

- Simple and efficient (garbage collection done as the program runs)
- Requires the compiler to generate the code to correctly track references.
- In reference cycles (e.g A references B which references A, even when no-one references them (they should both be collected) they keep eachother alive) blocks may be kept live indefinitely, the **GC** needs to avoid this.

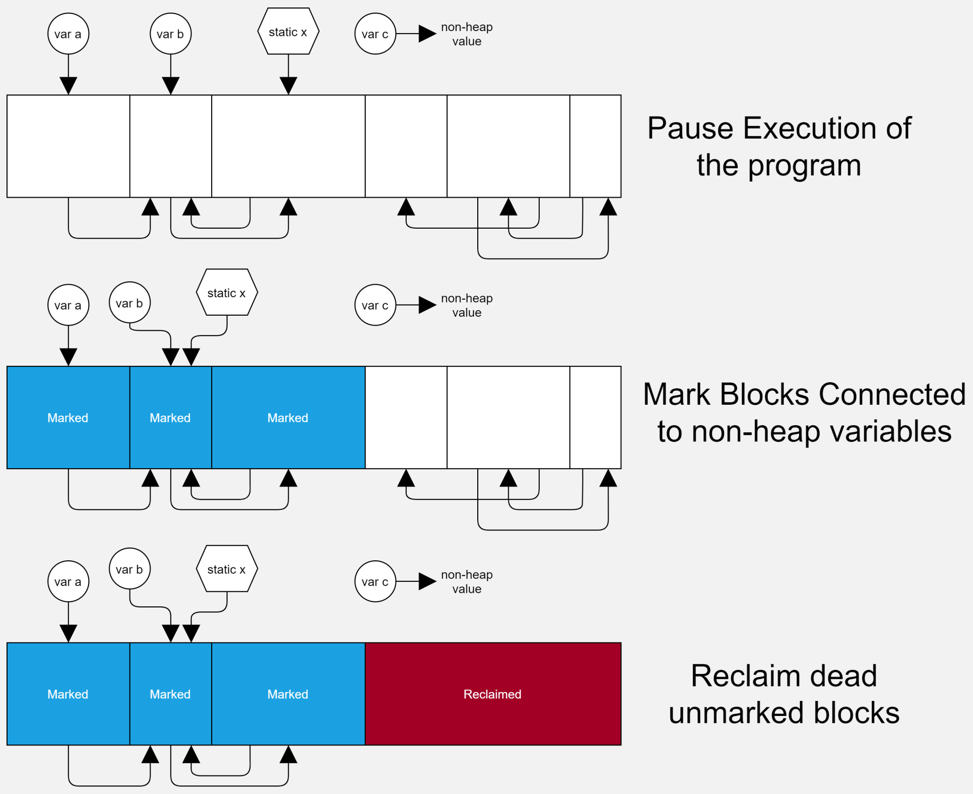
```python
# used for assigments such as a = b
def assign(lval, rval):
    if rval.on_heap():
        rval.references += 1

    if lval.on_heap():
        lval.references -= 1
        if lval.references == 0:
            gc.reclaim(lval.block)

    lval.value = rval.value
```

Definition: Mark-Sweep Garbage Collector

Pause the program, and collect all blocks that are not pointed to (potentially indirectly) by a non-heap value (e.g local or static variable).

This will find all dead blocks/garbage (unlike a **reference counting GC**) however it has considerable overhead. Each block's maintenance/housekeeping information must also contain space for a mark bit.

- **Mark** For all stack variables, if they point to the heap then recursively traverse, marking each block they point to as live.
- **Sweep** Go through each block, if it is not marked as live, then collect and deallocate it.


var a
var b
static x
var c
non-heap value
Pause Execution of the program
Marked
Mark Blocks Connected to non-heap variables
Reclaimed
Reclaim dead unmarked blocks

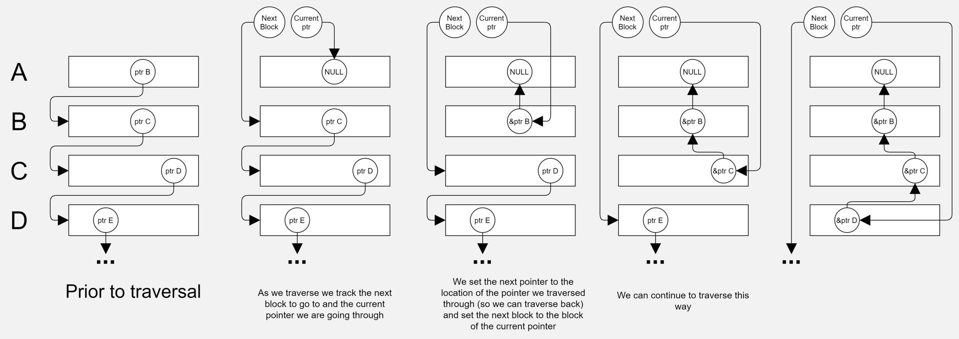
Definition: Pointer-Reversal Marking

Recursive traversal of pointers requires a potentially large amount of stack space (e.g going through a linked list of throusands of elements).

A garbage collector needs to work when memory is low. Hence we can use **Pointer-Reversal Marking** to store the pointers to return to (previous in traversal) in place.

1. When visiting a block, we select a pointer from within the block. We then set that pointer as the current, and place the address of the previous current pointer into it (hence now reversed).
2. We can then use the current pointer value to traverse to the next block, and continue.
3. When we find an already marked block, or no pointer in the block top traverse to, we can retreat back by using the pointer held in the location of the "current pointer" as this points back to the parent. We can set this
4. Note we need to store the number of marked children in the block's maintenance/-housekeeping data.

By storing the previously visited in the pointer we are traversing through, we require no extra stack space.


A
B
C
D
ptr B
ptr C
ptr D
ptr E
Next Block
Current ptr
NULL
&ptr B
&ptr C
&ptr D
Prior to traversal
As we traverse we track the next block to go to and the current pointer we are going through
We set the next pointer to the location of the pointer we traversed through (so we can traverse back) and set the next block to the block of the current pointer
We can continue to traverse this way

```c
// Variables used to keep track of location
void **current_ptr;
void **next_ptr;
void* current_block;

// traverse downwards
next_ptr = find_ptr_in_block(current_block);

current_block = *next_ptr; // current_block = address of next block
*next_ptr = current_ptr; // next_ptr now contains the address of where the old current_ptr was stored
current_ptr = next_ptr;

// we can mark the block we are current at
mark_block(current_block);

// traversing upwards (assuming no next_ptr)
next_ptr = *current_ptr; // the next pointer is now previous (traversing back)
*current_ptr = current_block; // set the current ptr back to being the next block
current_ptr = next_ptr; // current pointer moves to the previous
current_block = block_from_ptr(*current_ptr); //get the block we are now in
```

Definition: Two Space Garbage Collector

The heap is split into two, a **from-space** and a **two-space**. Blocks are allocated from the **from-space**. When there are no more free blocks, all live (reachable from a non-heap pointer) blocks are copied to the **to-space** and then the **to-space** becomes the **from-space** and vice-versa.

- Fast, there are few complex pointer manipulations.
- Automatically compacts memory when copying.
- Can place linked blocks close together in memory when copying to allow for better cache locality.

Definition: Generational Garbage Collector

The heap is split into several areas based on the age of blocks.

- Allocate new blocks from the youngest, if full, move the oldest young blocks to an older area.
- **GC** is used more frequently on the younger areas.
- Can apply different **GC** techniques to the different areas.

Java 13

Java 13 has 5 different garbage collectors, you can read more about there here.

- Serial
- Parallel
- Concurrent Mark-Sweep (CMS)
- Garbage First (G1),
- ZGC

Other Compiler Components

Definition: Debugger

Useful tools for inspecting the behaviour of programs.

- Program behaviour must be maintained when being debugged (execution time can change, but correctness must be consistent).
- Debugging information (e.g function and variable names, source line mappings) may be embedded in the program.
- Highly optimising compilers and debuggers are mostly incompatible (as a useful debugger can map behaviour back to source code, and no such relation exists once complex optimisations are done). Hence debuggers usually compile with optimisations off.

There are two main kinds of debuggers:

- **Interactive** e.g GDB Allow users to inspect the state of the program, variables, functions, memory etc and to pause execution of the program.

  For natively compiled (e.g Rust, C, Go) the architecture and OS must support breakpoints. For bytecode compiled (e.g Java) the interpreter supports this.
- **Post-Mortem / Core Dump** e.g Rust backtrace Provides debugging information upon failure by doing a reverse lookup from the program counter.
  - Stack TraceBack shows the methods called, their local variables, arguments etc.
  - Contents of global and dynamic/heap variables.

  This requires the debugger to work out where variables are stored (register, stack, heap) and to map them back to names from the source using debugging information embedded at compile time.

Definition: Profilers

Provide performance information on a program.

- Typically used once code is correct, algorithmically optimal and optimised by the compiler.
- Can determine the time spent in functions, or how much of a program is spent is certain sections of code.

For example a profiler could interrupt execution periodically (some number of ms), identify the method being used, and increment a counter for it. Hence after profiling we can get a breakdown of the proportion of program time spent in each function.