# Revision History

| Revision | Date | Author(s) | Description |
|---|---|---|---|
| 0.1 | 2017-05-02 | Martin | ROOPL++ design and heap design discussion |
| 0.2 | 2017-05-16 | Martin | Updated front-page logo and expanded heap design section |
| 0.3 | 2017-05-24 | Martin | Added Fragmentation and Garbage subsection. Added assumption about available heap manipulation subroutines for heap layout section. Added explanations of listings. Expanded Buddy-memory `get_free` algorithm with early idea for heap splitting/merging and heap growth |
| 0.4 | 2017-06-24 | Martin | Extended grammar in section 2.1. Added sections 2.2 to 2.9. Addressed Chapter 3 feedback. Rewrote pseudo-code algorithms in EXTENDED JANUS. |

CHAPTER 1

# Introduction

TODO

## 1.1 Reversible Computing

TODO

## 1.2 Reversible Object-Oriented Programming

The high-level reversible language Roopl (Reversible Object-Oriented Programming Language) was introduced in late 2016 and is, to our knowledge, the first of its kind [4]. The language extends the design of previously existing reversible imperative languages with object-oriented programming language features such as user-defined data types, class inheritance and subtype-polymorphism. As a first, Roopl successfully integrates the object-oriented programming (OOP) paradigms into the reversible computation setting using a static memory manager to maintain garbage-free computation, but at cost of "programmer usability" as objects only lives within `construct` / `deconstruct` blocks, which needs to be predefined, as the program call stacks is required to be reset before program termination.

## 1.3 Motivation

Roopl's block defined objects and lack of references are problematic when write complex, reversible programs using OOP methodologies. It has therefore been proposed to extend and partially redesign the language with dynamic memory management in mind, such that these shortcomings can be addressed, ultimately increasing the usability of reversible OOP. Work within the field of reversible computing related to heap manipulation [1], reference counting [9] and garbage collection [10] suggests that a Roopl extension is feasible.

## 1.4 Thesis Statement

TBD

## 1.5 Outline

TBC

CHAPTER 2

# The Roolp++ Language

With the design and implementation of the Reversible Object-Oriented Programming Language (Roopl), the first steps into the uncharted lands of Object-Oriented Programming (OOP) and reversibility was taken. In this chapter, we will present Roopl++ , the natural successor of Roopl, improving the language's object instantiation by letting objects live outside `construct / deconstruct` blocks, allowing complex, reversible programs to be written using OOP methodologies. As with its predecessor, Roopl++ is purely reversible and each component of a program written in Roopl++ is locally invertible. This ensures no computation history is required nor added program size for backwards direction program execution.

Inspired by other language successors such as C++ was to C, Roopl++ is a superset of Roopl, containing all original functionality of its predecessor, extended with new object instantiation methods for increased programming usability and an array type.

```
TODO
```

**Figure 2.1:** Example Roopl++ program

## 2.1 Syntax

A Roopl++ program consists, analogously to a Roopl program, of one or more class definitions, each with a varying number of fields and class methods. The program's entry point is a nullary main method, defined exactly once and is instantiated during program start-up. Fields of the main object will serve as output of the program, exactly as in Roopl.

**Roopl++ Grammar**

| | | | |
|---|---|---|---|
| $prog$ | $::=$ | $cl^+$ | (program) |
| $cl$ | $::=$ | $\textbf{class}\ c\ (\textbf{inherits}\ c)^?\ (t\ x)^*\ m^+$ | (class definition) |
| $d$ | $::=$ | $c \mid \textbf{vector}\!<\!t\!>$ | (class and array) |
| $t$ | $::=$ | $\textbf{int} \mid c \mid \textbf{vector}\!<\!t\!>$ | (data type) |
| $m$ | $::=$ | $\textbf{method}\ q\texttt{(}t\ x\texttt{,}\ \ldots\texttt{,}\ t\ x\texttt{)}\ s$ | (method) |
| $s$ | $::=$ | $x\ \odot\texttt{=}\ e \mid x\ \texttt{<=>}\ x \mid x[e]\ \odot\texttt{=}\ e$ | (assignment) |
| | $\mid$ | $\textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s\ \textbf{fi}\ e$ | (conditional) |
| | $\mid$ | $\textbf{from}\ e\ \textbf{do}\ s\ \textbf{loop}\ s\ \textbf{until}\ e$ | (loop) |
| | $\mid$ | $\textbf{construct}\ d\ x\quad s\quad \textbf{destruct}\ x$ | (object block) |
| | $\mid$ | $\textbf{local}\ t\ x = e\quad s\quad \textbf{delocal}\ t\ x = e$ | (local variable block) |
| | $\mid$ | $\textbf{new}\ d\ x \mid \textbf{delete}\ d\ x$ | (object con- and destruction) |
| | $\mid$ | $\textbf{call}\ q\texttt{(}x\texttt{,}\ \ldots\texttt{,}\ x\texttt{)} \mid \textbf{uncall}\ q\texttt{(}x\texttt{,}\ \ldots\texttt{,}\ x\texttt{)}$ | (local method invocation) |
| | $\mid$ | $\textbf{call}\ x\texttt{::}q\texttt{(}x\texttt{,}\ \ldots\texttt{,}\ x\texttt{)} \mid \textbf{uncall}\ x\texttt{::}q\texttt{(}x\texttt{,}\ \ldots\texttt{,}\ x\texttt{)}$ | (method invocation) |
| | $\mid$ | $\textbf{skip} \mid s\ s$ | (statement sequence) |
| $e$ | $::=$ | $\overline{n} \mid x \mid x[e] \mid \texttt{nil} \mid e\ \otimes\ e$ | (expression) |
| $\odot$ | $::=$ | $\texttt{+} \mid \texttt{-} \mid \texttt{\^{}}$ | (operator) |
| $\otimes$ | $::=$ | $\odot \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{\textbar} \mid \texttt{\&\&} \mid \texttt{\textbar\textbar} \mid \texttt{<} \mid \texttt{>} \mid \texttt{=} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=}$ | (operator) |

**Syntax Domains**

| | | |
|---|---|---|
| $prog \in$ Programs | $s \in$ Statements | $n \in$ Constants |
| $cl \in$ Classes | $e \in$ Expressions | $x \in$ VarIDs |
| $t \in$ Types | $\odot \in$ ModOps | $q \in$ MethodIDs |
| $m \in$ Methods | $\otimes \in$ Operators | $c \in$ ClassIDs |

**Figure 2.2:** Syntax domains and EBNF grammar for Roopl++

The Roopl++ grammar extends Roopl's grammar with a new dynamic array type in form of **vector**s and a new object lifetime option in form of working with objects outside of blocks, using the **new** and **delete** approach. Furthermore, the local block extension proposed in [4] has become a standard part of the language. Class definitions remains unchanged, an consists of a **class** keyword followed by a class name. Subclasses must be specified using the **inherits** keyword and a following parent class name. Classes can have any number of fields of any of the data types, including the new vector type. A class definition is required to include at least one method, defined by the **method** keyword followed by a a method name, a comma-separated list of parameters and a body.

Reversible assignments for integer variables and vector elements uses similar syntax as Janus assignments, by updating a variable through any of the addition (+=), subtraction or bitwise XOR operators. As with Janus, when updating a variable $x$ using any of said operators, the right-hand side of the operator argument must be entirely independent of $x$ to maintain reversibility. Usage of these reversible assignment operators for object or vector variables are undefined.

Roopl++ objects can be instantiated in two ways. Either using object blocks known from Roopl, or by using the **new** statement. The object-blocks have a limited lifetime, as the object only exists within the **construct** and **destruct** segments. Using **new** allows the object to live until program termination, if the program terminates with a **delete** call. By design, it is the programmers responsibility to deallocate objects instantiated by the usage of **new**.

The methodologies for argument aliasing and its restrictions on method on invocations from Roopl carries over in Roopl++ and object fields are as such disallowed as arguments to local methods to prevent irreversible updates and non-local method calls to a passed objects are prohibited. The parameter passing scheme remains call-by-reference and the Roopl's object model remains unchanged in Roopl++ .

## 2.2 Object Instantiation

Object instantiation through the **new** statement, follows the pattern of the mechanics of the **construct**/**destruct** blocks from Roopl, except for the less limited scoping and lifetime of objects. The mechanisms of the statement

$$\textbf{construct}\ c\ x \quad s \quad \textbf{destruct}\ x$$

are as follows:

1. Memory for an object of class $c$ is allocated. All fields are automatically zero-initialized by virtue of residing in already zero-cleared memory.
2. The block statement $s$ is executed, with the name $x$ representing a reference to the newly allocated object.
3. The reference $x$ may be modified by swapping its value with that of other references of the same type, but it should be restored to its original value within the statement block $s$, otherwise the meaning of the object block is undefined.
4. Any state that is accumulated within the object should be cleared or uncomputed before the end of the statement is reached, otherwise the meaning of the object block is undefined.
5. The zero-cleared memory is reclaimed by the system.

The statement pair consisting of

$$\textbf{new}\ c\ x \quad s \quad \textbf{delete}\ c\ x$$

could be considered an *fluid* block, meaning we can have overlapping blocks. We can as such initialize an object $x$ of class $c$ and an object $y$ of class $d$ and destroy $x$ before we destroy $y$, a feature that was not possible in ROOPL. The mechanisms of the **new** statement are as follows

1. Memory for an object of class $c$ is allocated. All fields are automatically zero-initialized by virtue of residing in already zero-cleared memory.
2. The block statement $s$ is executed, with the name $x$ representing a reference to the newly allocated object.

and the mechanisms of the **delete** statement are as follow

1. The reference $x$ may be modified by swapping its value with that of other references of the same type, but it should be restored to its original value before a **delete** statement is called on $x$, otherwise the meaning of the object deletion is undefined.
2. Any state that is accumulated within the object should be cleared or uncomputed before the **delete** statement is executed, otherwise the meaning of the object block is undefined.
3. The zero-cleared memory is reclaimed by the system.

The mechanisms of the **new** and **delete** statements are, essentially, a split of the mechanisms of the **construct**/**destruct** blocks into two separate statements. As with ROOPL, fields must be zero-cleared after object deletion, otherwise it is impossible for the system to reclaim the memory reversibly. This is the responsibility of the of the programmer to maintain this, and to ensure that objects are indeed deleted in the first place. A **new** statement without a corresponding **delete** statement targeting the same object further ahead in the program is undefined.

## 2.3 Vector Model

Besides asymmetric object lifetimes, ROOPL++ also introduces reversible, dynamic arrays in form of the **vector** type. While ROOPL on featured integers and custom data types in form of classes, one of its main inspirations, JANUS, implemented static, reversible arrays [**rg:janus**].

While ROOPL by design did not include any data storage language constructs, as they are not especially noteworthy nor interesting from an OOP perspective, they do generally improve the expressiveness of the language. For ROOPL++ these were decided to be part of the core language as array, especially multi-dimensional, dynamic arrays become interesting from a memory management perspective.

ROOPL++ 's dynamic arrays expand upon the array model from JANUS. Arrays are index by integers, starting from 0. In JANUS, only integer arrays were allowed and arrays were one-dimensional. In ROOPL++ arrays of any type can be defined, meaning either integer arrays, custom data types in form of class array or from other array types and have as such potential to be multi-dimensional.

Array element accessing is permitted using the bracket notation expression presented in Janus. Accessing an out-of-bounds index is undefined. Vector instantiation and element assignments, aliasing and circularity is described in detail in the following section.

Vectors can contain elements of different classes if, say class $A$ and $B$ both inherit from some class $C$ and vector $x$ is a **vector**<$C$> type. In this case, the array can hold elements of type $A$, $B$, and $C$.

## 2.4 Vector Instantiation

Vector instantiation uses the **new** and **delete** keywords to reversibly construct and destruct vector types. In irreversible OOP languages, instantiation and insertion/deletion of elements are usually executed in the the following steps for empty, dynamic arrays

1. A small amount of memory is reserved for the overhead of an array
2. Once the array is non-empty, a pointer for the overhead place in memory points to the beginning of the data.
3. As elements are inserted, the data memory is expanded to fit new elements, if we need indexes higher than the current size of the array.
4. As the highest indexed elements are deleted, the data memory is collapsed to reduce the overall size of the array object.

Consider the statement

**new vector<int>** $x$

in which we reserve memory for an **int vector** with the mechanics following mechanics

1. A small amount of memory is reserved for the overhead of an array.
2. On the first element insert, e.g. $x[n]$ += 1, a block of memory suitable for storing $|[0, 1, ..., n]|$ integer values are reserved, with the newly updated value of $x[n]$ inserted and uninitialized cells zero-cleared, and a pointer from the overhead to the data is linked.
3. If a value is inserted at, say, $x[n + 1]$, a similar action is dispatched, but deallocating the old block after allocating the new one first.
4. When the highest indexed cell element is zero-cleared, a allocation action is dispatched for obtaining a smaller block for our array data, while the old, larger block is freed.

In Roopl++ , we only allow instantiation of empty, dynamic arrays. As shown above, Integer elements are assigned using one of the reversible assignment operators. For class **vector**s and **vector vectors**, we assign cell elements a little differently. We make use of the **new** and **delete** statements, but instead of specifying which variable should hold the newly created/deleted object or array, we specify which array cell it should be stored in. The mechanisms of these assignments are equal to the example for **int vector**s; we grow the array on assignments to higher-than-current-max indexes and we shrink on zero-clearing of highest-indexed element.

`TODO: Code example of assigning` **`new`** $c$ $x[3]$ `to empty array`

As with ROOPL++ objects instantiated outside of **construct**/**destruct** blocks, **vector**s must be deleted before program termination to reversibly allow the system to reclaim the memory. Before deletion of a vector, all its elements must be zero-cleared such that the array is shrunk to the empty array.

Consider the statement

$$\textbf{delete vector<int>}\ x$$

with the following mechanics

1. The reference $x$ may be modified by swapping, assigning cell element values and zero-clearing cell element values, but must be restored to a 0-element sized array before the **delete** statement. Otherwise, the meaning of the statement is undefined.
2. If the reference $x$ is the empty array upon the **delete** statement execution, the zero-cleared memory is reclaimed by the system.

With reversible, dynamic arrays of varying types and dimensionality, we must be extremely careful when updating and assigning values, to ensure we maintain reversibility and avoid irreversible statements. Therefore, when assigning or updating integer elements with one of the reversible assignment operators, we prohibit the cell value from being reference on the right hand side, meaning the following statement is prohibited

$$x[5]\text{+=}\ x[5] + 1$$

However, we do allow other, initialized (non-zero-cleared array elements) to be referenced in the right hand side of the statement.

## 2.5 Local blocks

## 2.6 Type System

## 2.7 Language Semantics

## 2.8 Program Inversion

## 2.9 Computational Strength

CHAPTER 3

# Compilation

The following chapter will present the design for the reversible heap manager and the translation schemes used in the process of translating ROOPL++ to the reversible low-level machine language PISA.

## 3.1 Dynamic Memory Management

Native support of complex data structures is not a trivial matter to implement in a reversible computing environment. Variable-sized records and frames need to be stored efficiently in a structured heap, while avoiding garbage build-up to maintain reversibility. A reversible heap manager layout has been proposed for a simplified version of the reversible functional language RFUN and later expanded to allow references to avoid deep copying values [1, 12, 9].

The following section presents a discussion of various heap manager layouts along with their advantages and disadvantages in terms of implementation difficulty, garbage build-up and the OOP domain.

### 3.1.1 Fragmentation

An important matter to consider when designing a heap layout for dynamic memory allocation is efficient memory usage. In a stack allocating memory layout, the stack discipline is in effect, meaning only the most recently allocated data can be freed. This is not the case with heap allocation, where data can be freed regardless of allocation order. This feature comes with the consequence of potential memory fragmentation, as blocks are being freed in any order [8].

We distinguish different types of fragmentation as internal or external fragmentation.

#### Internal Fragmentation

Internal fragmentation occurs in the memory heap when part of an allocated memory block is unused. This type of fragmentation can arise from a number of different scenarios, but mostly it originates from *over-allocating*.

An example of *over-allocating* would be a scenario where we are allocating memory for an object of size $m$ onto a simple, fixed-sized block heap, where the fixed block size is $n$ and $m \neq n$. If $n > m$, internal fragmentation would occur of size $n - m$ for every object of size $m$ allocated in said heap. If $n < m$, numerous blocks would be required for allocation to fit our object. In this case the internal fragmentation would be of size $m + n - (m + n) \mod n$ per allocated object of size $m$.

TODO

**Figure 3.1:** Example of internal fragmentation due to *over-allocation*

Figure 3.1 visualizes the examples of internal fragmentation build-up from *over-allocating* memory.

Intuitively, internal fragmentation can be prevented by ensuring that the size of the block(s) being used for allocating space for an object of size $m$ either match or sums to this exact size.

### External Fragmentation

External fragmentation materializes in the memory heap when a freed block becomes partly or completely unusable for future allocation if it, say, is surrounded by allocated blocks but the size of the freed block is too small to contain any object on its own.

This type of fragmentation is generally a more substantial cause of problems than internal fragmentation, as the size of external fragmentation blocks usually are larger than the ones created from internal fragmentation, depending on the heap implementation (In a layout using variable-sized blocks of, say, size $2^n$, the internal fragment size becomes considerable for large $n$s).

Non-allocatable external fragments become a problem when there it is impossible to allocate space for a large object as a result of too many non-consecutive blocks scattered around the heap, caused by the external fragmentation. Physically, there is enough space to store the object, but not in the current heap state. In this scenario we would need a garbage collector to clean the heap and relocate blocks in such a manner that the fragmentation disperses.

TODO

**Figure 3.2:** Example of external fragmentation

### Consequences of Fragmentation in the Reversible Setting

Fragmentation management and handling is particularly interesting within in the reversible setting. As reversible programmers we strive to maintain an *exact* (or equivalent) computational state during program execution, such that we at any time can reverse the computational direction and return the a previous state. If external memory fragmentation is left unchecked in a reversible dynamic memory allocation implementation, we can easily run into the following scenario

- Object $A$, $B$ and $C$ are allocated on the heap in that order. The size of $A$ is one block of memory, $B$' size is two blocks and $C$'s size is three blocks.
- $B$ is freed, leaving external fragmentation of two blocks in between $A$ and $C$.

- $C$ is freed, thus leaving a total of five blocks of freed memory after $A$.
- The computational direction is reverted and we now allocate $C$ again, which is allocated in the first three blocks next to $A$.
- $B$ is then allocated again, thus giving us the order $A - C - B$ instead of the original $A - B - C$ order.

Figure 3.3 shows this exact scenario.

TODO

**Figure 3.3:** Potential state-changing side effect of external fragmentation in the reversible setting

Furthermore, it is in our interest to limit external fragmentation as removing it during program execution would require a garbage collector, which is a non-trivial matter to implement. Mogensen presented a heap manager for reversible functional languages using reference counts and hashing to achieve garbage collection in [10], but it is not immediately clear how this garbage collection could translate to an object-oriented language.

### 3.1.2 Memory Garbage

Traditionally, freed memory blocks are simply re-added to the free-list during deallocation and no modification of the actual data stored in the block is done. In the reversible setting we must return the memory block to the original state the block was in before the allocation occurred (e.g. zero-cleared registers), to uphold the time-invertible and two-directional computational model.

Thus, in the reversible setting it should be our goal to also return the memory to its original state after program termination. With heap allocation, we will need one or more free-lists to keep track of free blocks during program execution, which must be stored in memory, besides the actual heap representation. These free-lists can essentially be considered garbage and as such, they must also be returned to their original state after execution. Furthermore, the if the heap grows during execution, it should be returned to its original size.

Returning the free-list(s) to their original states is a non-trivial matter, depending on the heap layout. Axelsen and Glück introduced heap allocation and deallocation, but without restoring the free-list to its original state in [1]. The argument for not restoring the free-list is that it can be considered harmless garbage in the sense that the free-list left after termination is equivalent, as it contains the same blocks, but linked in a different order, depending on the allocation and deallocation order during program execution.

This intuitively leads to the question of garbage classification. In the reversible setting all functions are injective. Thus, given some $input_f$, the injective function $f$ produces some $output_f$ and some $garbage_f$ (e.g. storing data in the heap, so the free list changes, the heap grows, etc.). Its inverse function $f^{-1}$ must thus take $f$'s $output_f$ and $garbage_f$ as $input_{f^{-1}}$ to produce its output $output_{f^{-1}}$ which is $f$'s $input_f$. However, in the context of reversible heaps, we must consider all free-lists as of "equivalent garbage class" and thus freely substitutable with each other, as injective functions still can drastically change the block layout, free-list order, etc. during its execution in either direction. Figure 3.4 shows how any free-list can be passed between a function $f$ and its inverse $f^{-1}$.

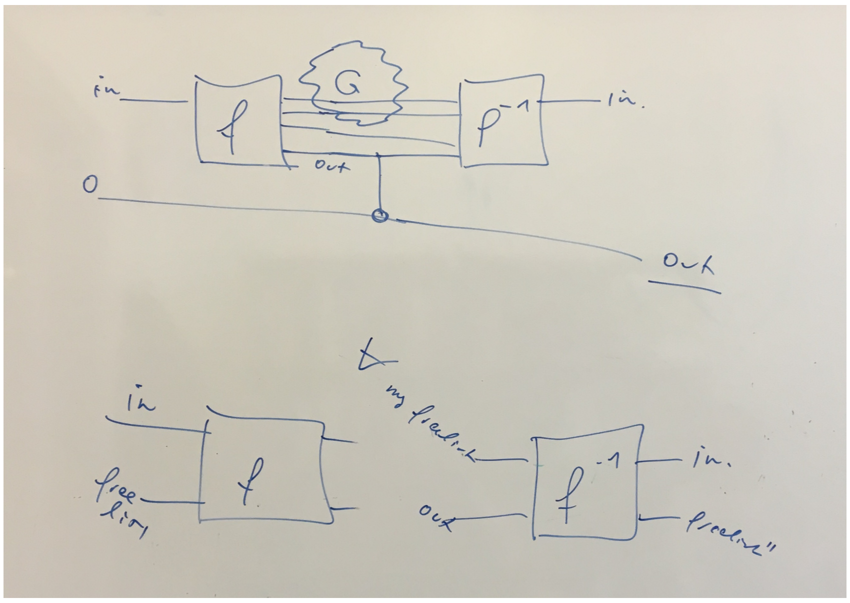

**Figure 3.4:** All free-lists are considered equivalent in terms of injective functions (Temporary photo)

### 3.1.3 Heap manager layouts

Heap managers can be implemented in numerous ways. Different layouts yield advantages when allocating memory, finding a free block or when collecting garbage. As our goal is to construct a garbage-free heap manager, our finalized design should emphasize and reflect this objective in particular. Furthermore, we should attempt to allocate and deallocate memory as efficiently as possible, as merging and splitting of blocks is a non-trivial problem in a reversible setting.

For the sake of simplicity, we will not consider the the issue of retrieving memory pages reversibly. A reversible operating system is a long-term dream of the reversible programmer and as reversible programming language designers, we assume that ROOPL++ will be running in a environment, in which the operating system will be supplying memory pages and their mappings. As such, the following heap memory designs reflect this preliminary assumption, that we always can query the operating system for more memory.

Historically, most object-oriented programming languages utilize a dynamic memory manager during program execution. In older, lower-level languages such as C, memory allocation had to be stated explicitly and with the requested size through the `malloc` statement and deallocated using the `free` statement. Modern languages, such as C++, JAVA and PYTHON, *automagically* allocates and frees space for objects and variable-sized arrays by utilizing their dynamic memory manager to dispatch `malloc`- and `free`-like operations to the operating system and managing the obtained memory blocks in private heap(s) [7, 11, 2]. The heap layout of these managers vary from language to language and compiler to compiler.

Previous work on reversible heap manipulation has been done for reversible functional languages in [1, 3, 10].

`TODO: Something about Cons/Nil heap from [1]`

TODO: Something about ref count extension [9]

For the sake of simplicity in the following heap layout pseudo-code outlines, we assume access to the following subroutines (with parameter passing), inspired by the body of Axelsen and Glück's get_free subroutine.

In order to reversibly allocate a block, we assume access to the subroutine allocate_block which, given the register $r_{cell}$ where we want the address of the allocated block stored and the register $r_{flp}$ containing the free-list pointer, effectively pops the head of the free-list to $r_{cell}$. Listing 3.1 shows the subroutine.

```
procedure allocate_block($r_{cell}$, $r_{flp}$)
    EXCH $r_{cell}$ $r_{flp}$
    SWAP $r_{cell}$ $r_{flp}$
```

**Listing 3.1:** allocate_block subroutine

In order to grow the heap, we assume access to the subroutine grow_heap which, given the register $r_{hp}$ containing the heap pointer and a number $n$ specifying how much the heap should grow, increments the heap pointer by $n$. Listing 3.2 shows the subroutine.

```
procedure grow_heap($r_{hp}$, $n$)
    ADDI $r_{hp}$ $n$
```

**Listing 3.2:** grow_heap subroutine

Finally, in order to move the heap, we assume access to the subroutine move_heap which, given the register $r_{hbp}$ containing a pointer to the *bottom of a heap*[1] and the register $r_{hp}$ containing the heap pointer and a number $n$ representing the amount the heap should be moved, iterates every memory address in the heap, starting at the top, and moves the memory block $n$ spaces. This is down using a top-down approach to avoid overwriting data.

```
procedure move_heap($r_{hbp}$, $r_{hp}$, $n$, $m$)
    from  i = $r_{hp}$
    do
        SWAP $r_i$ $r_{i+n}$
        i -= 1              ; 1 word (?)
    until i = r_{hbp} - n
```

**Listing 3.3:** move_heap subroutine

### Memory Pools

A simple layout for a heap manager would allocate memory using fixed-size blocks regardless of the actual size of the record, which is known as memory pooling [5]. The advantage of a heap layout following this approach would lie in the simplicity of its implementation, as a simple linked list of identical-sized free cells would need to be maintained. If we assume a our fixed-sized blocks are of size $n$ machine words, the following get_free subroutine allocates and deallocates memory cells for a record of size $m$.

[1]assuming it exists

```
if (r_flp == 0)
then
    XOR r_cell r_hp
    call grow_heap(r_hp, m + n − (m + n)%n)
else
    call allocate_block(r_cell, r_flp)
fi (r_flp == 0) && (r_cell == r_hp - (m + n − (m + n)%n))
```

**Listing 3.4:** Allocating and deallocating records of size $m$ using block of a fixed size $n$. Code modified from [1]

Listing 3.4 shows the modified `get_free` subroutine for allocating and deallocating memory blocks in a memory pool layout. If the free-list is empty, the block which the heap pointer is currently pointing at is allocated and the heap is grown by $m + n - (m + n)\%n$ blocks (rounding up to nearest number of blocks of size $n$ when allocating object of size $m$). If the free-list is non-empty, the `allocate_block` subroutine is called.

A huge disadvantage to using fixed-sized memory blocks is the external fragmentation that occurs when freeing blocks, if the program's objects are not of the same size. When freeing a number of blocks in the middle of a section of allocated blocks, external fragmentation occurs, which becomes a problem if we need to allocate space for a large record but only have small sections of fixed-blocks available, scattered throughout the heap. A garbage collector could solve this issue, but is a non-trivial matter to implement.

`TODO: Graphics`

`TODO: Present various layout using fixed-size blocks - Trees, continuous?`

### Buddy Memory

In this approach we have all blocks be variable-size of the power-of-two. Having different block sizes and methods for splitting a larger block into smaller ones reduces fragmentation. In terms of implementation, reversible splitting and merging might be possible as we're always doubling the size when merging and halving when splitting, thus making the operations logically inverse of each other [6].

We can reversibly and recursively split blocks using subroutine `split_block`. The subroutine takes two registers $r_{cell1}$ and $r_{cell2}$ to contain the split block. If the passed register $r_{fpl}$ containing the free-list pointer of the heap for $2^{n+1}$ is empty, we recursively call the `split_block`[2]. If the free-list of the heap for $2^{n+1}$ heap is not empty, we pop the head of the free list into $r_{cell1}$ and split the block by setting the pointer halfway through the block in register $r_{cell2}$. Listing 3.5 shows the subroutine.

```
procedure split_block(r_cell1, r_cell2, r_fpl, n)
    if (r_flp == 0)
    then
        call split_block(r_cell1, r_cell2, r_fpl, n * 2)
    else
        EXCH r_cell1 r_flp
        SWAP r_cell1 r_flp
        EXCH r_cell2 r_cell1 + n / 2
    fi (r_flp == 0)
```

[2]Unsure if this works?

**Listing 3.5:** `split_block` subroutine

`TODO: Consider Torben's suggesting with only merging/splitting if adjacent block`

Assuming we have access to an array of pointers $flps[]$, pointing to the beginning of a free list, index at its power-of-two and subroutine in Listing 3.5, we could allocate and deallocate a record of size $n$ in a buddy memory layout reversibly in the following manner

```
maxPo2 = 2^10
from powerOfTwo = 2
do
    powerOfTwo = powerOfTwo * 2
until powerOfTwo < n

if (flps[powerOfTwo] == 0)
then
    from nextPo2 = powerOfTwo * 2
    do
        call split_block($r_{cell}^{powerOfTwo}$, $r_{cell2}^{powerOfTwo}$, flps[nextPo2], nextPo2)
    until (nextPo2 > maxPo2)

     if (flps[powerOfTwo] == 0)
     then
        call grow_heap(r_{hp}^{maxPo2}, maxPo2)
        call split_block($r_{cell}^{powerOfTwo}$, $r_{cell2}^{powerOfTwo}$, flps[nextPo2], nextPo2)
    else
        EXCH flps[powerOfTwo] $r_{cell2}^{powerOfTwo}$
    fi (flps[powerOfTwo] == 0)
else
    call allocate_block($r_{cell}^{powerOfTwo}$, flps[powerOfTwo])
fi (flps[powerOfTwo] == 0)
```

**Listing 3.6:** Allocating and deallocating records of size $n$ using Buddy Memory

`TODO: Consider need_max_heap_grow subroutine?`

The `get_free` modification for a buddy memory layout starts by finding the $2^n$ heap which the record of size $n$ can fit it. Afterwards, if the free-list pointer for the found heap is empty, the `split_block` subroutine is called to obtain two blocks of the given size. If no blocks can be found and split (recursively), the biggest $2^n$ heap is grown by one block and the then `split_block` is called again (recursively) to obtain two blocks of the desired size. Once two blocks have been obtained, the first is used for allocating our object and the second is set as the head of the free list. In the case we the free list of the found heap is non-empty, we simply call the `allocate_block` subroutine.

`TODO: Graphics`

### One heap per record type

Another layout approach would be maintaining one heap per record type in the program. During compilation, classes would be analysed and a heap for each class would be created. The advantage of this approach would be less garbage generation, as each allocation is tailored as closely as possible to the size of the record obtained from a static analysis during compilation. The obvious disadvantage is the amount of book-keeping and workload associated with growing and shrinking a heap and its neighbours.

```
Algorithm:  Determine heap for record type, call get_free on found heap
```

```
TODO: Graphics
```

### Shared heap, record type-specific free lists

A combination of the previous layouts would consist of a single heap, but record-specific free lists. This way, we ensure minimal fragmentation when allocating and freeing as the different free lists ensures that allocation of an object wastes as little memory as possible. By only keeping one heap, we eliminate the growth/shrinking issues of the multiple heap layout. There is, however, still a considerable amount of book-keeping involved in maintaining multiple free-lists. The bigger the number of unique classes defined in a program, the more free-lists we need to maintain during execution. If the free-lists are not allowed to point at the same free block (which they intuitively shouldn't in order to ensure reversibility), programs with many classes, say one hundred, could potentially fill up the heap with one hundred free lists, but only ever allocate objects for one of the classes, thus wasting a lot of memory on unused free lists. This scenario could potentially be avoided with compiler optimizations.

```
Algorithm:  Find block size for record type-specific free list, call get_free
on shared heap
```

```
TODO: Graphics
```

### One heap per power-of-two

A different approach as to having one heap per record type, would be having one heap per power-of-two until some arbitrary size. Using this approach, records would be stored in the heap which has a block size of a power-of-two closes matching to the record's size. This layout is a distinction from the "one heap per record type" as it still retains the size-optimized storing idiom but allows the heaps to contain records of mixed types. For programs with a large amount of small, simple classes needed to model some system, where each class is roughly the same size, the amount of heaps constructed would be substantially smaller than using one heap per record type, as many records will fit within the same heap. Implementation wise, the number of heaps can be determined at compile time. Furthermore, to ensure we do not end up with heaps of very large memory blocks, an arbitrary power-of-two size limit could be set at, say, 1 kb . If any record exceeds said limit, it could be split into $\sqrt{n}$ size chunks and stored in their respective heaps. This approach does, however, also suffer from large amount of book-keeping and fiddling when shrinking and growing adjacent heaps.

```
Algorithm:  Similar to buddy memory?
```

```
TODO: Graphics
```

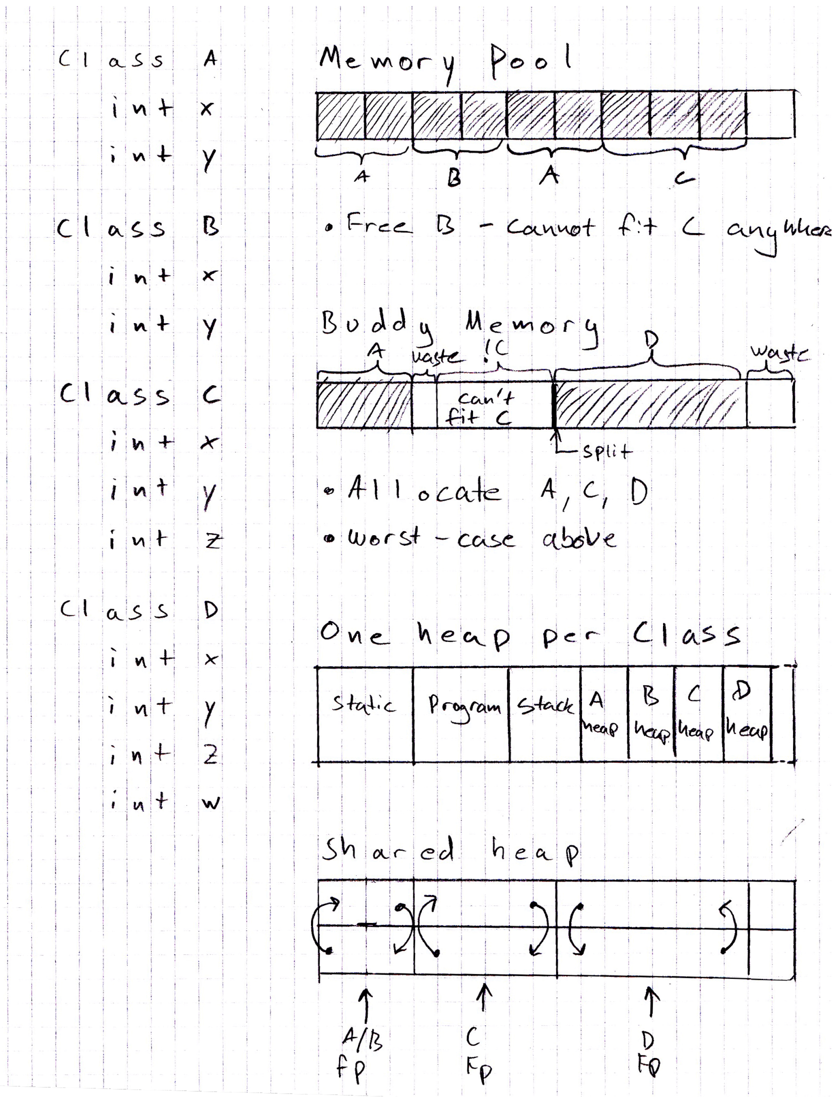

**Figure 3.5:** Heap memory layouts (Draft)

## 3.2 Roopl++ Memory Layout

TBD