Bab 13

Runtime Environment dan Memory Management

13.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep runtime environment dan perannya dalam eksekusi program
2. Menjelaskan struktur activation record (stack frame) dan komponen-komponennya
3. Mengimplementasikan simulator runtime stack untuk function calls
4. Memahami memory layout: static, stack, dan heap
5. Menjelaskan mekanisme heap management dan garbage collection
6. Mengimplementasikan manajemen memory untuk runtime environment sederhana

13.2 Pendahuluan

Runtime environment adalah konteks di mana program yang telah dikompilasi dieksekusi. Menurut sumber dari StudyLib:

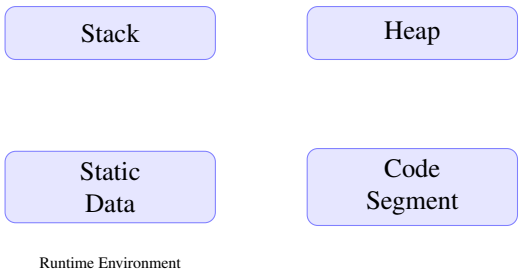
> "Runtime environment: stack, heap, activation records; garbage collection intro. Managing run-time structures (activation records, memory layout, symbol tables)."[1]

Runtime environment mencakup:

- **Memory Organization**: Bagaimana memory diorganisir dan dialokasikan untuk berbagai jenis data
- **Calling Conventions**: Mekanisme pemanggilan fungsi, passing parameter, dan return values

- **Scope Management**: Bagaimana variabel diakses berdasarkan scope-nya (local, non-local, global)
- **Memory Management**: Alokasi dan dealokasi memory untuk variabel dan data structures

Gambar 13.1 menunjukkan komponen-komponen runtime environment.



Gambar 13.1: Komponen-komponen runtime environment

Asumsi runtime untuk proyek compiler subset C (stack/activation record, layout variabel) dibahas di Bagian 13.8 dan dipakai oleh code generation (Bab 14).

Runtime environment harus dirancang dengan hati-hati karena mempengaruhi:

- Efisiensi eksekusi program
- Keamanan memory (memory safety)
- Kemampuan mendukung fitur bahasa (recursion, nested functions, closures, dll.)
- Portabilitas antar platform

13.3 Memory Layout

Program yang dieksekusi memiliki memory layout yang terorganisir menjadi beberapa region. Setiap region memiliki karakteristik dan tujuan penggunaan yang berbeda.

13.3.1 Memory Regions

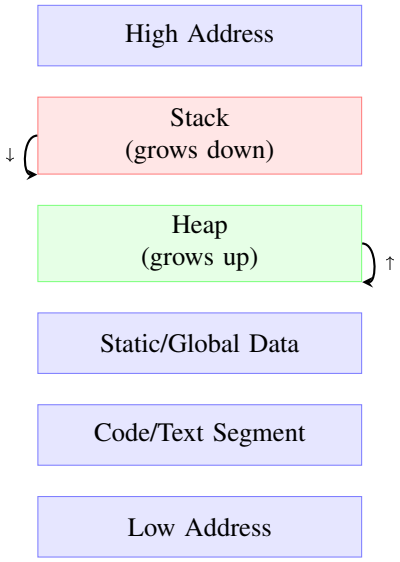
Memory address space program biasanya dibagi menjadi beberapa region utama:

1. **Code/Text Segment**: Berisi instruksi machine code yang dihasilkan compiler. Region ini biasanya read-only dan tidak dapat dimodifikasi saat runtime.
2. **Static/Global Data**: Berisi variabel global dan static yang dialokasikan pada compile time. Region ini memiliki ukuran tetap dan alamat yang diketahui saat compile time.

3. **Stack**: Region untuk activation records (stack frames) dari fungsi-fungsi yang sedang aktif. Stack tumbuh ke bawah (dari high address ke low address) dan dikelola secara otomatis.

4. **Heap**: Region untuk dynamic memory allocation. Heap tumbuh ke atas (dari low address ke high address) dan dikelola secara manual atau melalui garbage collector.

Gambar 13.2 menunjukkan layout memory secara visual dengan TikZ.



Gambar 13.2: Memory layout program

Gambar 13.3 menunjukkan layout memory yang khas:

```
High Address
    +------------------+
    |   Command Line   |
    |     Arguments    |
    +------------------+
    |       Stack      | <- Tumbuh ke bawah
    |    (grows down)  |
    |                  |
    |        ↓         |
    |                  |
    |        ↑         |
    |    (grows up)    |
    |       Heap       | <- Tumbuh ke atas
    +------------------+
    |  BSS (uninit)    |
    |  Data (init)     |
    +------------------+
    |  Text/Code       | <- Read-only
Low Address
```

Gambar 13.3: Memory layout khas untuk program yang dieksekusi

13.3.2 Static Memory Allocation

Static memory allocation terjadi pada compile time. Variabel yang dialokasikan secara static memiliki:

- Alamat yang tetap dan diketahui saat compile time
- Lifetime yang sama dengan program (dari awal hingga akhir eksekusi)
- Tidak memerlukan runtime overhead untuk alokasi/dealokasi

Contoh variabel static:

- Variabel global: `int global_var;`
- Variabel static lokal: `static int counter;`
- String literals dan konstanta

Keuntungan static allocation:

- Sangat efisien (tidak ada overhead runtime)
- Deterministik (alamat diketahui saat compile time)
- Tidak ada risiko memory leak

Keterbatasan:

- Tidak mendukung recursion dengan baik
- Ukuran harus diketahui saat compile time
- Tidak fleksibel untuk dynamic data structures

13.3.3 Stack-Based Memory Allocation

Stack digunakan untuk activation records dari fungsi-fungsi yang sedang aktif. Stack allocation memiliki karakteristik:

- **Automatic**: Alokasi dan dealokasi terjadi otomatis saat fungsi dipanggil dan kembali
- **LIFO**: Last In First Out - fungsi terakhir dipanggil adalah yang pertama kembali
- **Fast**: Alokasi/dealokasi sangat cepat (hanya mengubah stack pointer)
- **Limited Lifetime**: Data di stack hanya hidup selama fungsi aktif

Stack sangat cocok untuk:

- Local variables
- Function parameters
- Return addresses
- Temporary values

Contoh penggunaan stack:

Listing 13.1: Contoh program yang menggunakan stack

```
int factorial(int n) {
    if (n <= 1) return 1;
    int temp = n * factorial(n - 1);  // Recursive call
    return temp;
}

int main() {
    int result = factorial(5);  // Stack frames untuk main dan factorial
    return 0;
}
```

13.3.4 Heap-Based Memory Allocation

Heap digunakan untuk dynamic memory allocation yang tidak dapat ditangani oleh stack. Heap allocation memiliki karakteristik:

- **Manual Management**: Programmer harus secara eksplisit mengalokasikan dan membebaskan memory
- **Flexible Lifetime**: Object di heap dapat hidup lebih lama dari fungsi yang membuatnya
- **Variable Size**: Ukuran dapat ditentukan saat runtime
- **Slower**: Alokasi/dealokasi lebih lambat dibanding stack

Heap digunakan untuk:

- Dynamic arrays dan data structures
- Objects yang harus hidup lebih lama dari fungsi pembuatnya
- Shared data structures
- Large objects yang tidak muat di stack

Contoh penggunaan heap:

Listing 13.2: Contoh penggunaan heap

```
int* createArray(int size) {
    int* arr = new int[size];  // Alokasi di heap
    return arr;  // Pointer ke heap, valid setelah fungsi kembali
}

void useArray() {
    int* myArray = createArray(100);
    // Gunakan array...
    delete[] myArray;  // Dealokasi manual
}
```

13.4 Activation Records (Stack Frames)

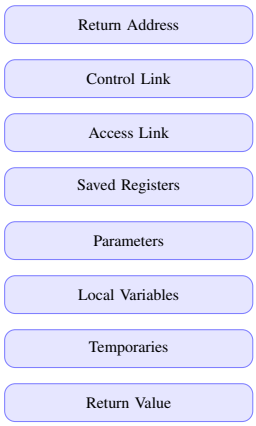
Activation record (juga disebut stack frame) adalah struktur data yang digunakan untuk menyimpan informasi tentang eksekusi satu fungsi. Setiap kali fungsi dipanggil, activation record baru dibuat di stack.

13.4.1 Komponen Activation Record

Activation record biasanya berisi komponen-komponen berikut:

1. **Return Address**: Alamat instruksi di caller yang harus dieksekusi setelah fungsi kembali
2. **Control Link (Dynamic Link)**: Pointer ke activation record dari caller (fungsi yang memanggil)
3. **Access Link (Static Link)**: Pointer ke activation record dari enclosing scope (untuk nested functions)
4. **Saved Registers**: Nilai register yang harus disimpan dan dikembalikan setelah fungsi selesai
5. **Parameters**: Nilai parameter yang diteruskan ke fungsi (actual parameters)
6. **Local Variables**: Variabel lokal yang dideklarasikan dalam fungsi
7. **Temporary Values**: Nilai sementara yang digunakan selama komputasi dalam fungsi
8. **Return Value**: Nilai yang dikembalikan fungsi (jika ada)

Gambar 13.4 menunjukkan struktur activation record secara visual.
Gambar 13.5 menunjukkan struktur activation record yang khas:



Gambar 13.4: Struktur activation record

```
High Address
    +----------------------+
    |  Return Address      |
    +----------------------+
    |  Control Link (FP)   | -> Activation record caller
    +----------------------+
    |  Access Link         | -> Enclosing scope (if nested)
    +----------------------+
    |  Saved Registers     |
    +----------------------+
    |  Parameters          |
    |    param1            |
    |    param2            |
    +----------------------+
    |  Local Variables     |
    |    local1            |
    |    local2            |
    +----------------------+
    |  Temporaries         |
    |    temp1             |
    |    temp2             |
    +----------------------+
    |  Return Value        |
Low Address
```

Gambar 13.5: Struktur activation record (stack frame)

13.4.2 Calling Sequence

Calling sequence adalah urutan instruksi yang dihasilkan compiler untuk memanggil fungsi. Terdapat dua bagian:

Caller Sequence (Prologue)

Instruksi yang dijalankan oleh caller sebelum memanggil fungsi:

1. Evaluasi actual parameters (dari kanan ke kiri atau kiri ke kanan, tergantung calling convention)
2. Push parameters ke stack (atau pass melalui register)
3. Save caller-saved registers
4. Push return address
5. Transfer control ke callee (CALL instruction)

Callee Sequence (Prologue)

Instruksi yang dijalankan oleh callee di awal fungsi:

1. Save frame pointer (FP) dari caller
2. Set FP baru ke current stack pointer (SP)
3. Allocate space untuk local variables (adjust SP)
4. Save callee-saved registers (jika diperlukan)

Return Sequence (Epilogue)

Instruksi yang dijalankan saat fungsi kembali:

1. Place return value (di register atau stack)
2. Restore callee-saved registers
3. Restore SP (deallocate local variables)
4. Restore FP dari control link
5. Restore return address
6. Return control ke caller (RET instruction)

13.4.3 Contoh Calling Sequence

Mari kita lihat contoh calling sequence untuk program sederhana:

Listing 13.3: Contoh program untuk analisis calling sequence

```c
int add(int x, int y) {
    int sum = x + y;
    return sum;
}

int main() {
    int a = 5;
    int b = 10;
    int result = add(a, b);
    return 0;
}
```

Assembly code yang dihasilkan (simplified):

```assembly
main:
    push rbp                ; Save caller's frame pointer
    mov rbp, rsp            ; Set new frame pointer
    sub rsp, 16             ; Allocate space for locals (a, b, result)

    mov [rbp-4], 5          ; a = 5
    mov [rbp-8], 10         ; b = 10

    ; Call add(a, b)
    mov eax, [rbp-8]        ; Load b
    push eax                ; Push parameter 2
    mov eax, [rbp-4]        ; Load a
    push eax                ; Push parameter 1
    call add                ; Call function
    add rsp, 8              ; Clean up parameters
    mov [rbp-12], eax       ; result = return value

    mov eax, 0              ; return 0
    mov rsp, rbp            ; Restore stack pointer
    pop rbp                  ; Restore frame pointer
    ret                      ; Return

add:
    push rbp                ; Save caller's frame pointer
    mov rbp, rsp            ; Set new frame pointer
    sub rsp, 4              ; Allocate space for local (sum)

    mov eax, [rbp+8]        ; Load x (parameter 1)
    add eax, [rbp+12]       ; Add y (parameter 2)
    mov [rbp-4], eax        ; sum = x + y
    mov eax, [rbp-4]        ; Load sum for return

    mov rsp, rbp            ; Restore stack pointer
    pop rbp                  ; Restore frame pointer
    ret                      ; Return
```

13.5 Implementasi Runtime Stack Simulator

Untuk memahami runtime stack dengan lebih baik, kita akan mengimplementasikan simulator sederhana dalam C++.

13.5.1 Struktur Data Activation Record

Listing 13.4: Struktur data untuk activation record

```cpp
#include <string>
#include <vector>
#include <unordered_map>
#include <iostream>

// Representasi activation record
struct ActivationRecord {
    std::string function_name;           // Nama fungsi
    void* return_address;                // Return address (simulated)
    ActivationRecord* control_link;      // Pointer ke caller's AR
    ActivationRecord* access_link;       // Pointer ke enclosing scope

    // Local variables dan parameters
    std::unordered_map<std::string, int> locals;
    std::unordered_map<std::string, int> parameters;

    // Return value
    int return_value;

    ActivationRecord(const std::string& name,
                     ActivationRecord* caller = nullptr)
        : function_name(name),
          return_address(nullptr),
          control_link(caller),
          access_link(nullptr),
          return_value(0) {}
};
```

13.5.2 Stack Manager

Listing 13.5: Implementasi runtime stack manager

```cpp
class RuntimeStack {
private:
    ActivationRecord* top;  // Top of stack (current activation)
    int frame_count;

public:
    RuntimeStack() : top(nullptr), frame_count(0) {}

    // Push activation record baru (function call)
    void pushFrame(const std::string& function_name) {
        ActivationRecord* new_frame =
```

```
            new ActivationRecord(function_name, top);
        new_frame->access_link = top;  // Simplified: same as control
↪ link
        top = new_frame;
        frame_count++;

        std::cout << ">>> Called: " << function_name
                  << " (Frame #" << frame_count << ")\n";
        printStack();
    }

    // Pop activation record (function return)
    void popFrame() {
        if (top == nullptr) {
            std::cerr << "Error: Cannot pop from empty stack!\n";
            return;
        }

        std::string func_name = top->function_name;
        int ret_val = top->return_value;

        ActivationRecord* old_top = top;
        top = top->control_link;
        delete old_top;
        frame_count--;

        std::cout << "<<< Returned from: " << func_name
                  << " (return value: " << ret_val << ")\n";
        printStack();
    }

    // Get current activation record
    ActivationRecord* getCurrentFrame() {
        return top;
    }

    // Set local variable di current frame
    void setLocal(const std::string& name, int value) {
        if (top == nullptr) {
            std::cerr << "Error: No active frame!\n";
            return;
        }
        top->locals[name] = value;
        std::cout << "  Set local: " << name << " = " << value << "\n";
    }

    // Get local variable dari current frame atau enclosing scopes
    int getLocal(const std::string& name) {
        ActivationRecord* frame = top;
        while (frame != nullptr) {
            if (frame->locals.find(name) != frame->locals.end()) {
                return frame->locals[name];
            }
            frame = frame->access_link;  // Check enclosing scope
```

```
        }
        std::cerr << "Error: Variable '" << name
                  << "' not found!\n";
        return 0;
    }

    // Set parameter
    void setParameter(const std::string& name, int value) {
        if (top == nullptr) {
            std::cerr << "Error: No active frame!\n";
            return;
        }
        top->parameters[name] = value;
        std::cout << "  Set parameter: " << name << " = " << value << "\n
↪ ";
    }

    // Set return value
    void setReturnValue(int value) {
        if (top == nullptr) {
            std::cerr << "Error: No active frame!\n";
            return;
        }
        top->return_value = value;
        std::cout << "  Set return value: " << value << "\n";
    }

    // Print stack untuk debugging
    void printStack() {
        std::cout << "Stack (top to bottom):\n";
        ActivationRecord* frame = top;
        int level = 0;
        while (frame != nullptr) {
            std::cout << "  [" << level << "] "
                      << frame->function_name << "\n";
            frame = frame->control_link;
            level++;
        }
        std::cout << "\n";
    }

    ~RuntimeStack() {
        while (top != nullptr) {
            popFrame();
        }
    }
};
```

13.5.3 Contoh Penggunaan Simulator

Listing 13.6: Contoh penggunaan runtime stack simulator

```
int main() {
```

```
    RuntimeStack stack;

    // Simulasi: main() calls factorial(5)
    stack.pushFrame("main");
    stack.setLocal("n", 5);

    // Call factorial(5)
    stack.pushFrame("factorial");
    stack.setParameter("n", 5);

    // Recursive call: factorial(4)
    stack.pushFrame("factorial");
    stack.setParameter("n", 4);

    // Recursive call: factorial(3)
    stack.pushFrame("factorial");
    stack.setParameter("n", 3);

    // Base case: factorial(1) returns 1
    stack.setReturnValue(1);
    stack.popFrame();

    // factorial(3) = 3 * factorial(2) = 3 * 2 = 6
    // (simplified, actual would need more frames)
    stack.setReturnValue(6);
    stack.popFrame();

    stack.setReturnValue(24);
    stack.popFrame();

    stack.setReturnValue(120);
    stack.popFrame();

    // main returns
    stack.popFrame();

    return 0;
}
```

13.6 Heap Memory Management

Heap memory management adalah proses mengalokasikan dan membebaskan memory di heap secara dinamis. Terdapat dua pendekatan utama:

13.6.1 Manual Memory Management

Dalam manual memory management (seperti C/C++), programmer harus secara eksplisit:

- Mengalokasikan memory: `malloc()`, `new`
- Membebaskan memory: `free()`, `delete`

Keuntungan:

- Kontrol penuh atas memory
- Tidak ada overhead garbage collector
- Predictable performance

Kekurangan:

- Rentan terhadap memory leaks
- Dangling pointers
- Double free errors
- Memory fragmentation

13.6.2 Allocation Algorithms

Heap manager menggunakan berbagai algoritma untuk mengalokasikan memory:

First Fit

Mencari block pertama yang cukup besar:

- Cepat (tidak perlu mencari semua)
- Dapat menyebabkan fragmentation

Best Fit

Mencari block terkecil yang cukup besar:

- Mengurangi wasted space
- Lebih lambat (harus mencari semua)
- Dapat menyebabkan banyak small fragments

Worst Fit

Mencari block terbesar:

- Meninggalkan large free blocks
- Dapat mengurangi fragmentation kecil

Buddy Allocation

Membagi memory menjadi blocks dengan ukuran power of 2:

- Mudah untuk merge adjacent blocks
- Dapat menyebabkan internal fragmentation

13.6.3 Garbage Collection

Garbage collection adalah automatic memory management yang membebaskan memory yang tidak lagi digunakan. Menurut sumber dari StudyLib:

> "Runtime environment: stack, heap, activation records; garbage collection intro. Managing run-time structures (activation records, memory layout, symbol tables)."[1]

Konsep Garbage Collection

Garbage collector mengidentifikasi dan membebaskan memory yang tidak lagi dapat diakses (unreachable) dari program. Object dianggap garbage jika:

- Tidak ada pointer/reference yang menunjuk ke object tersebut
- Tidak dapat diakses dari root set (stack, global variables, registers)

Strategi Garbage Collection

Mark and Sweep

1. **Mark Phase**: Traverse dari root set, mark semua reachable objects
2. **Sweep Phase**: Scan semua objects, free yang tidak di-mark

Keuntungan:

- Dapat menangani cyclic references
- Tidak memerlukan memory compaction

Kekurangan:

- Dapat menyebabkan fragmentation
- Stop-the-world pauses

Reference Counting Setiap object memiliki counter yang menghitung jumlah reference ke object tersebut. Ketika counter menjadi 0, object di-free.

Keuntungan:

- Incremental (tidak perlu stop-the-world)
- Memory dibebaskan segera saat tidak digunakan

Kekurangan:

- Tidak dapat menangani cyclic references
- Overhead untuk setiap assignment

Copying Collector (Generational GC) Memory dibagi menjadi young generation dan old generation. Young objects yang survive beberapa collections dipromote ke old generation.

Keuntungan:

- Efisien untuk short-lived objects
- Automatic compaction

Kekurangan:

- Memerlukan extra memory (copying)
- Overhead untuk promotion

Implementasi Sederhana Mark and Sweep

Berikut adalah implementasi sederhana mark-and-sweep garbage collector:

Listing 13.7: Implementasi sederhana mark-and-sweep GC

```
#include <vector>
#include <unordered_set>

class GCObject {
public:
    bool marked;
    std::vector<GCObject*> references;

    GCObject() : marked(false) {}
    virtual ~GCObject() {}

    void addReference(GCObject* obj) {
        references.push_back(obj);
    }
};

class SimpleGC {
```

```
private:
    std::vector<GCObject*> heap;
    std::vector<GCObject*> roots;  // Root set (stack, globals)

public:
    // Allocate new object
    GCObject* allocate() {
        GCObject* obj = new GCObject();
        heap.push_back(obj);
        return obj;
    }

    // Add to root set
    void addRoot(GCObject* obj) {
        roots.push_back(obj);
    }

    // Mark phase: mark all reachable objects
    void mark() {
        std::vector<GCObject*> worklist = roots;

        while (!worklist.empty()) {
            GCObject* obj = worklist.back();
            worklist.pop_back();

            if (!obj->marked) {
                obj->marked = true;
                // Add all references to worklist
                for (GCObject* ref : obj->references) {
                    if (!ref->marked) {
                        worklist.push_back(ref);
                    }
                }
            }
        }
    }

    // Sweep phase: free unmarked objects
    void sweep() {
        auto it = heap.begin();
        while (it != heap.end()) {
            GCObject* obj = *it;
            if (!obj->marked) {
                delete obj;
                it = heap.erase(it);
            } else {
                obj->marked = false;  // Reset for next collection
                ++it;
            }
        }
    }

    // Run garbage collection
    void collect() {
```

```
        mark();
        sweep();
    }

    ~SimpleGC() {
        for (GCObject* obj : heap) {
            delete obj;
        }
    }
};
```

13.7 Memory Layout untuk Program Contoh

Mari kita analisis memory layout untuk program yang lebih kompleks:

Listing 13.8: Program contoh untuk analisis memory layout

```
int global_var = 100;           // Static/Global
static int static_var = 200;    // Static

int* createArray(int size) {    // Function
    int* arr = new int[size];   // Heap allocation
    return arr;
}

int factorial(int n) {          // Function
    static int counter = 0;     // Static local
    counter++;

    if (n <= 1) return 1;
    int temp = n * factorial(n - 1);  // Stack: recursive
    return temp;
}

int main() {                    // Function
    int local_a = 10;           // Stack: local variable
    int local_b = 20;           // Stack: local variable

    int* heap_array = createArray(100);  // Heap allocation

    int result = factorial(5);  // Stack: recursive calls

    delete[] heap_array;        // Heap deallocation
    return 0;
}
```

Memory layout saat eksekusi:

```
High Address
    +---------------------+
    |  Stack (grows down) |
    |                     |
    |  [factorial(1) AR]  | <- Top of stack
```

```
    |    n = 1             |
    |    temp = ?          |
    |    return addr       |
    +----------------------+
    |  [factorial(2) AR]   |
    |    n = 2             |
    |    temp = ?          |
    +----------------------+
    |  ...                 |
    +----------------------+
    |  [factorial(5) AR]   |
    |    n = 5             |
    |    temp = ?          |
    +----------------------+
    |  [main AR]           |
    |    local_a = 10      |
    |    local_b = 20      |
    |    heap_array = ptr  |
    |    result = ?        |
    +----------------------+
    |                      |
    |  (free space)        |
    |                      |
    +----------------------+
    |  Heap (grows up)     |
    |  [heap_array[100]]   | <- 100 integers
    +----------------------+
    |  BSS                 |
    |  (uninitialized)     |
    +----------------------+
    |  Data (initialized)  |
    |    global_var = 100  |
    |    static_var = 200  |
    |    counter = ?       |
    +----------------------+
    |  Text/Code           |
    |  (instructions)      |
Low Address
```

13.8 Runtime Proyek Subset C

Untuk compiler proyek subset C (Bab 1–12), asumsi runtime disederhanakan: program berupa barisan statement (tanpa fungsi tambahan di fase awal), sehingga tidak ada pemanggilan fungsi pengguna. Variabel `int`/`float` dialokasikan secara statis atau pada stack dengan layout tetap; code generator (Bab 14) memakai offset/alamat dari

symbol table. Jika kelak proyek diperluas dengan fungsi, calling convention dapat mengikuti konvensi cdecl (stack-based arguments, caller-saved/callee-saved registers) agar konsisten dengan runtime environment yang dibahas di bab ini. Rincian layout activation record dan rencana calling convention untuk proyek dicatat dalam dokumentasi folder `proyek-compiler-subset-c/` agar Bab 14 memiliki acuan saat menghasilkan kode.

13.9 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Runtime environment adalah konteks eksekusi program yang mencakup memory organization, calling conventions, dan memory management
2. Memory layout terdiri dari code segment, static/global data, stack, dan heap, masing-masing dengan karakteristik dan tujuan penggunaan yang berbeda
3. Activation records (stack frames) menyimpan informasi tentang eksekusi fungsi, termasuk parameters, local variables, return address, dan links
4. Stack-based allocation cocok untuk local variables dengan automatic management, sementara heap allocation diperlukan untuk dynamic data dengan flexible lifetime
5. Garbage collection adalah teknik automatic memory management yang membebaskan unreachable objects, dengan berbagai strategi seperti mark-and-sweep, reference counting, dan generational GC

Runtime proyek subset C (layout variabel, rencana calling convention bila diperluas) memberi acuan untuk code generation di Bab 14.

Pemahaman tentang runtime environment dan memory management sangat penting untuk:

- Merancang compiler yang efisien
- Memahami bagaimana program dieksekusi
- Mengoptimalkan penggunaan memory
- Mengimplementasikan fitur bahasa seperti recursion, closures, dan dynamic allocation

13.10 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang runtime environment dan memory management, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [2] - Bab 7: Run-Time Environments
- **Engineering a Compiler**: Cooper & Torczon (2011) [3] - Bab 6: The Procedure Abstraction
- **StudyLib - Outcomes-Based Education**: Materials tentang runtime environment dan activation records [1]
- **UC San Diego CSE 231**: Course materials tentang compiler construction dan runtime organization [4]
- **Northeastern University CS 4410**: Comprehensive compiler design course dengan coverage runtime issues [5]

Daftar Pustaka

[1] StudyLib. *Outcomes-Based Education Curricula*. Materials on runtime environment and activation records. 2024. URL: https://studylib.net/doc/14111770/outcomes-based-education-curricula--academic-year-2015-.

[2] Alfred V. Aho **andothers**. *Compilers: Principles, Techniques, and Tools*. 2nd. Dragon Book. Pearson Education, 2006.

[3] Keith D. Cooper **and** Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2011.

[4] UC San Diego CSE Department. *CSE 231: Compiler Construction / Advanced Compiler Design*. Course materials and syllabus. 2024. URL: https://catalog.ucsd.edu/archive/2024-25/courses/CSE.html.

[5] Northeastern University. *CS 4410/6410: Compiler Design*. Course syllabus and materials, taught by Benjamin Lerner. 2024. URL: https://course.ccs.neu.edu/cs4410sp25/.