Run Time Environment

- From Lecture 1 to 11, we have seen many jobs that are done by a compiler.
- In addition to those jobs, compiler does another important job, which is called creating and managing run time environment.
- Run time environment means:
  1. Allocation and deallocation of memory for the variables, objects, and data
  2. To create mechanism for target program to access the variables, objects, and data
  3. Linkage between functions and procedures
  4. The mechanism for passing parameters between function and procedures
  5. Interfaces to input/output devices, operating systems, and other programs
- In this lecture, we only see 1 and 2. These two steps are done by memory management
Memory Management

- Compiler only assumes run physical memory, which is RAM (not other memories, like hard disk)
- Compiler divides this memory as follows (this is one possible way).
- The addressing of the memory that is created/used by the compiler is *logical addressing*. Actual addressing of memory is done by OS, which is called *physical addressing*.

---

**Memory Management**

- Memory is divided into two regions: *static region and dynamic region*
- *Static region* is divided into two parts: code and static
- **Code**: Target machine codes (for example, RISC assembly codes that we have seen in Lecture 11) are stored here. This area of memory is static (does not change). Because after compilation, the target program is generated and does not change anymore during the run.
- **Static**: In this area of memory, compiler stores program data that do not change through out the running of the program. For example, global constants, global variables.
- Compiler tries to put as many data as possible to the static region. Because, the addresses of the data in this region can be computed during compile time, not in the run time. So, running will be faster.
- Allocation in this region is called *static allocation*.
Memory Management

- **Dynamic region** is divided into two parts: stack and heap. Their size can be changed during the program execution. Compiler does not know during compile time how much memory may be required. These are decided during run time.
- Allocation in this region is called *dynamic allocation*
- **Stack**:
  - Stack grows towards bottom to top
  - Stack stores all local variables that are declared in a function, including the local parameters. Memory for these variables are created during run time. After the function execution is finished, they are deleted from the stack
  - Stack also stores status of the machine during function call, such as value of program counter, value of the registers. When the program comes back from the function, these values are restored

- **Heap**:
  - The remaining free space in dynamic region is heap
  - Heap grows from top to bottom
  - Stack and heap grow in two opposite way so that they do not overlap until the memory is available
  - Memory of this region can be allocated by the programmers themselves
  - For example, in C++, there is a `new` operator that creates a new object. This object is stored in heap by C++ compiler
  - For example, in C there are `malloc()` function that can be used by the programmer. This allocates memory from heap
  - Memory allocated for variable in one function can live even after the function is returned and the variable can be used in another function
  - When the allocated memory is no more used, the programmer can make that memory free. Or, the compiler can free it when it sees that the use of the memory is finished
  - During allocation and freeing of memory, many small small spaces are scattered in the memory. Compiler collects them all and make them a single big free space. This is called *garbage collection*
C++ Example 1

```c
#include<stdio.h>
#include<iostream>
using namespace std;

int const size = 20;
char *my_name;
int my_age, my_grades[10];

void print_my_name(void)
{
    int i;
    my_name = new char[size];
    my_name = "Masud Hasan";
    for(i=0; i<size; i++)
        cout << my_name[i];
}

int main(int)
{
    print_my_name();
    delete [] my_name;
}
```

- Equivalent assembly code goes to **code**
- Constant: it goes to **static**.
- Global variables: go to **static**. `my_name` is only a pointer here, actual array content goes to stack, see below.
- Local variable: `i` goes to **stack**
- **new** operator allocates memory of size 20 for `my_name` from **heap**
- Function call: program counter, registrar values are stored in **stack**. After returning from the function, these values are freed from stack
- Delete will delete the memory from heap that was allocated for `my_name`

C++ Example 1 (Continued)

```
#include<stdio.h>
#include<iostream>
using namespace std;

int const size = 20;
char *my_name;
int my_age, my_grades[10];

void print_my_name(void)
{
    int i;
    my_name = new char[size];
    my_name = "Masud Hasan";
    for(i=0; i<size; i++)
        cout << my_name[i];
}

int main(int)
{
    print_my_name();
    delete [] my_name;
}
```

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Assembly Code</strong></td>
</tr>
<tr>
<td>LD ...., ....</td>
</tr>
<tr>
<td>ST ...., ....</td>
</tr>
<tr>
<td>ADD ...., ....</td>
</tr>
<tr>
<td><strong>Static</strong></td>
</tr>
<tr>
<td>size, my_name, age</td>
</tr>
<tr>
<td>my_grade[10]</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>Program Counter, Registrar during call to <code>print_my_name()</code></td>
</tr>
<tr>
<td><strong>Heap</strong></td>
</tr>
<tr>
<td>my_name[20]</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>my_name deleted from heap</td>
</tr>
</tbody>
</table>
Stack Allocation (in detail)

• All programming language that use functions (similarly, methods, procedures) manage part of dynamic memory for stack allocation
• For example, C has functions, Pascal has functions and procedures, Java has methods. All of them use stack allocation
• When a function \( f_1 \) is called, it is called the activation of \( f_1 \). If another function \( f_2 \) is called from \( f_1 \), then \( f_2 \) is activated, before the function \( f_1 \) terminates.
• Similarly, in recursive function calls, many copies of same function are activated at the same time
• Relations among all activated functions can be represented by a tree, which is called activation tree
• Once a function is called, its local variables (including local parameters) are pushed into the stack. When the function terminates, they are popped off the stack.

Example 2 (Quick Sort)

```c
int a[10];
void readArray() { /* Reads 9 integers into a[1],...,a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value \( v \), and partitions \( a[m..n] \) so that
    \( a[m..p-1] \) are less than \( v \), \( a[p] = v \), and \( a[p+1..n] \) are
    equal to or greater than \( v \). Returns \( p \). */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (m > n) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = 99999;
    a[10] = 99999;
    quicksort(1,9);
}
```

This example takes 9 integers in an array as input and sorts them by quick sort (remember: quick sort is a recursive function).
This example takes 9 integers in an array as input and sorts them by quick sort (remember: quick sort is a recursive function).

Look at these two i. They are **not same**, because they are local to two different functions.

Similarly, look at these two sets of m and n. They are **not same**, because they are local to two different functions.

---

**Example 2 (Quick Sort)**

```c
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
} int partition(int m, int n) {
    /* Picks a separator value e, and partitions a[m..n] so that a[m..p-1] are less than e, a[p] = e, and a[p+1..n] are equal to or greater than e. Returns p. */
    ...
} void quicksort(int m, int n) {
    int i;
    if (m < n) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
} main() {
    readArray();
a[0] = -9999;
a[10] = 9999;
    quicksort(1,9);
}
```

Figure 7.2: Sketch of a quicksort program

---

**Example 2 (Remember: Quick Sort Partition)**

- Original array: 27 28 12 39 27 26
- First partition: 16 12 27 39 27 28 (Place pivot between S₁ and S₂)
Meaning of the tree nodes
m: main
r: readArray
q: quicksort
p: partition

Note that:
• This tree is only one possibility. For different partitioning, there will be different activation and different activation tree
• Activation starts from root, then from left to right, like preorder traversal

Execution sequence (red color): like preorder traversal
• Once the execution enters a function, the function becomes live
• When the execution exits the function, it is no more live
For each live function, compiler stores (push) an *activation record* in the stack.

Activation record contains the local variables of that function as well as other data local to that function.

Once a function execution is completed, its activation record is deleted (pop) from the stack.

--- means finished execution.