Compiler Design

UNIT V: CODE OPTIMIZATION

UNIT V: CODE OPTIMIZATION

Topics we are going to cover

1. Introduction– Principal Sources of Optimization
2. Optimization of basic Blocks
3. Loop Optimization
4. Runtime Environments – Source Language issues
5. Introduction to Global Data Flow Analysis
6. Storage Organization
7. Storage Allocation strategies – Access to non-local names
8. Parameter Passing.

CODE OPTIMIZATION

- Elimination of unnecessary instructions in object code, or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is usually called "code improvement" or "code optimization."

1 .The Principal Sources of Optimization

- A compiler optimization must preserve the semantics of the original program.
- Except in very special circumstances, once a programmer chooses and implements a particular algorithm, the compiler cannot understand enough about the program to replace it with a substantially different and more efficient algorithm.
- A compiler knows only how to apply relatively low-level semantic transformations, using general facts such as algebraic identities like i + 0 = i or program semantics such as the fact that performing the same operation on the same values yields the same result.

1 .The Principal Sources of Optimization

- Function – Preserving Transformations
  - Common Subexpressions
  - Copy Propagation
  - Dead-Code Elimination
  - Constant Folding
- Loop Optimization
  - Code Motion
  - Induction-Variable Elimination
  - Reduction in Strength

1 .The Principal Sources of Optimization

1. Causes of Redundancy
2. Global Common Subexpressions
3. Copy Propagation
4. Dead-Code Elimination
5. Code Motion
6. Induction Variables and Reduction in Strength

1.1 Causes of Redundancy

- There are many redundant operations in a typical program.
- Sometimes the redundancy is available at the source level.
- For instance, a programmer may find it more direct and convenient to recalculate some result, leaving it to the compiler to recognize that only one such calculation is necessary.
- But more often, the redundancy is a side effect of having written the program in a high-level language.

1.1 Causes of Redundancy

- In most languages (other than C or C++, where pointer arithmetic is allowed), programmers have no choice but to refer to elements of an array or fields in a structure through accesses like A[i][j] or X -> f 1.
- As a program is compiled, each of these high-level data-structure accesses expands into a number of low-level arithmetic operations, such as the computation of the location of the (i, j)th element of a matrix A.
- Accesses to the same data structure often share many common low-level operations. Programmers are not aware of these low-level operations and cannot eliminate the redundancies themselves.

1.1 Causes of Redundancy

- It is, in fact, preferable from a software-engineering perspective that programmers only access data elements by their high-level names; the programs are easier to write and, more importantly, easier to understand and evolve.
- By having a compiler eliminate the redundancies, we get the best of both worlds: the programs are both efficient and easy to maintain.

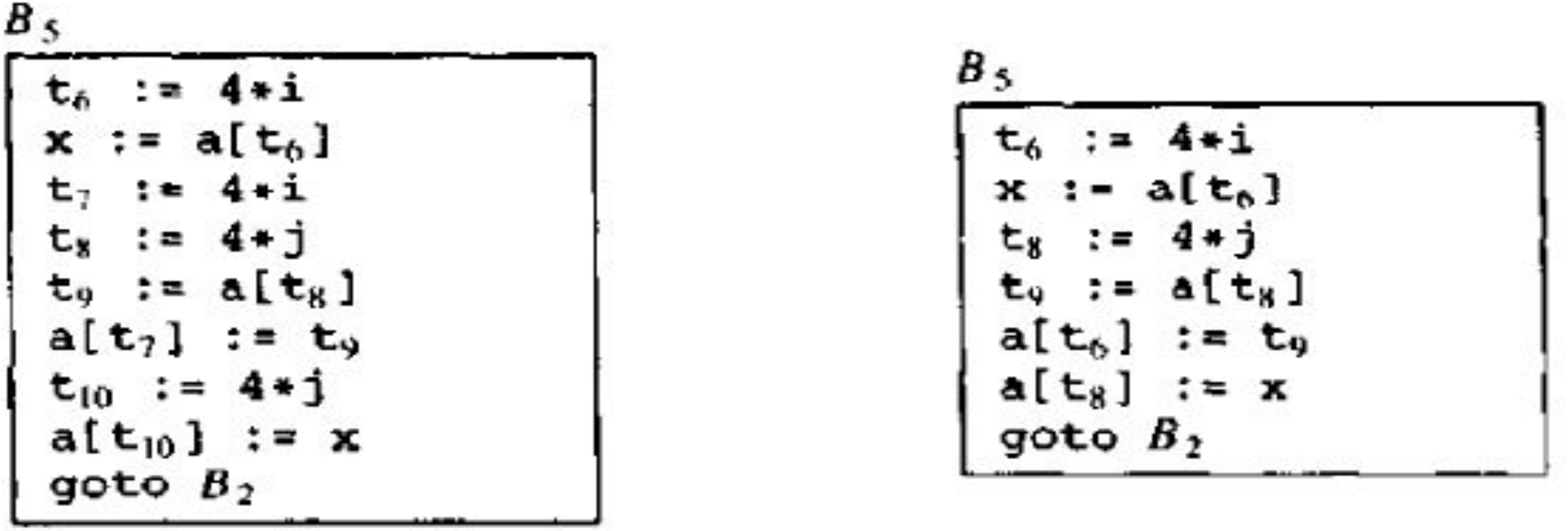
1 .The Principal Sources of Optimization

1. Causes of Redundancy
2. Global Common Subexpressions
3. Copy Propagation
4. Dead-Code Elimination
5. Code Motion
6. Induction Variables and Reduction in Strength

1.2 Global Common Subexpressions

- An occurrence of an expression E is called a *common subexpression* if *E* was previously computed and the values of the variables in E have not changed since the previous computation.
- We avoid recomputing E if we can use its previously computed value; that is, the variable x to which the previous computation of *E* was assigned has not changed in the interim.

1.2 Global Common Subexpressions



B5

```
t6  := 4*i
x  := a[t6]
t7  := 4*i
t8  := 4*j
t9  := a[t8]
a[t7] := t9
t10  := 4*j
a[t10] := x
goto B2
```

(a) Before

B5

```
t6  := 4*i
x  := a[t6]
t8  := 4*j
t9  := a[t8]
a[t6]  := t9
a[t8]  := x
goto B2
```

(b) After

Fig. 10.6. Local common subexpression elimination.

1 .The Principal Sources of Optimization

1. Causes of Redundancy
2. Global Common Subexpressions
3. Copy Propagation
4. Dead-Code Elimination
5. Code Motion
6. Induction Variables and Reduction in Strength

1.3 Copy Propagation

- copy propagation is the process of replacing the occurrences of targets of direct assignments with their values.
- A direct assignment is an instruction of the form x=y, which simply assigns the value of y to x. From the following code:

```
y = x
z = 3 + y
```

- Copy propagation would yield:

```
z = 3 + x
```

- Reduce the copying
- If y is reassigned in between, then this action cannot be performed

1 .The Principal Sources of Optimization

1. Causes of Redundancy
2. Global Common Subexpressions
3. Copy Propagation
4. Dead-Code Elimination
5. Code Motion
6. Induction Variables and Reduction in Strength

1.4 Dead Code Elimination

- A variable is *live* at a point in a program if its value can be used subsequently; otherwise, it is *dead* at that point.
- A related idea is *dead* (or *useless) code* - statements that compute values that never get used.
- While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

1.4 Dead Code Elimination

Example: Suppose debug is set to TRUE or FALSE at various points in the program, and used in statements like

```
if (debug) print ...
```

- It may be possible for the compiler to deduce that each time the program reaches this statement, the value of debug is FALSE. Usually, it is because there is one particular statement

```
debug = FALSE
```

that must be the last assignment to debug prior to any tests of the value of debug, no matter what sequence of branches the program actually takes.

- If copy propagation replaces debug by FALSE, then the print statement is dead because it cannot be reached.
- We can eliminate both the test and the print operation from the object code.
- More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as *constant folding.*

1.4 Dead Code Elimination

- One advantage of copy propagation is that it often turns the copy statement into dead code.

1 .The Principal Sources of Optimization

1. Causes of Redundancy
2. Global Common Subexpressions
3. Copy Propagation
4. Dead-Code Elimination
5. Code Motion
6. Induction Variables and Reduction in Strength

1.5 Code Motion

- Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time.
- The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

1.5 Code Motion

- An important modification that decreases the amount of code in a loop is *code motion.*
- This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a *loop-invariant computation)* and evaluates the expression before the loop.

1.5 Code Motion

- **Example** : Evaluation of *limit* - 2 is a loop-invariant computation in the following while-statement :

```
while (i <= limit-2)   /* statement does not change limit */
```

- Code motion will result in the equivalent code

```
t = limit-2
while ( i <= t ) /* statement does not change limit or t */
```

- Now, the computation of *limit - 2* is performed once, before we enter the loop.
- Previously, there would be n + 1 calculations of *limit - 2* if we iterated the body of the loop n times.

1 .The Principal Sources of Optimization

1. Causes of Redundancy
2. Global Common Subexpressions
3. Copy Propagation
4. Dead-Code Elimination
5. Code Motion
6. Induction Variables and Reduction in Strength

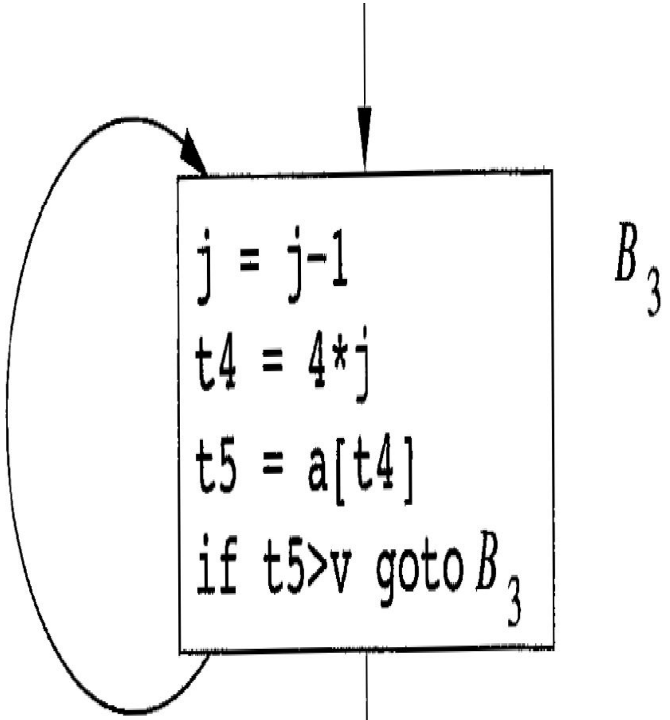
1.6 Induction Variables & Reduction in Strength

- Another important optimization is to find induction variables in loops and optimize their computation.
- A variable x is said to be an "induction variable“ if there is a positive or negative constant **c** such that each time x is assigned, its value increases by c.

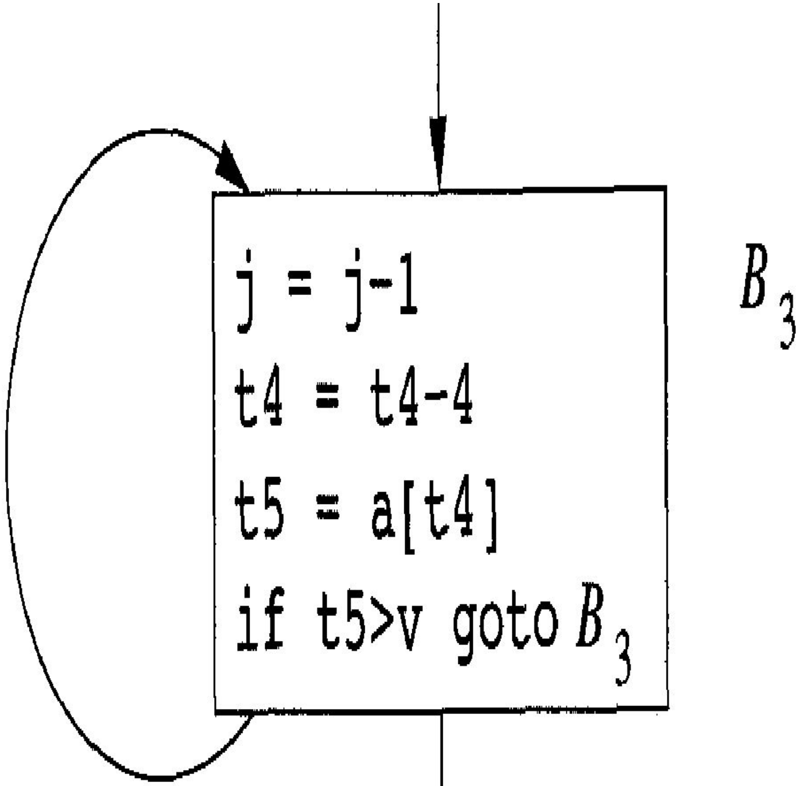
1.6 Induction Variables & Reduction in Strength

- Induction variables can be computed with a single increment (addition or subtraction) per loop iteration.
- The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as *strength reduction.*
- Induction variables allow us to perform a strength reduction.

1.6 Induction Variables & Reduction in Strength



```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```



```
j = j-1
t4 = t4-4
t5 = a[t4]
if t5>v goto B3
```

When processing loops, it is useful to work "inside-out" ; that is, we shall start with the inner loops and proceed to progressively larger, surrounding loops.

UNIT V: CODE OPTIMIZATION

Topics we are going to cover

1. Introduction– Principal Sources of Optimization
2. Optimization of basic Blocks
3. Loop Optimization
4. Runtime Environments – Source Language issues
5. Introduction to Global Data Flow Analysis
6. Storage Organization
7. Storage Allocation strategies – Access to non-local names
8. Parameter Passing.

3. Loop Optimization

- **loop optimization** is the process of increasing execution speed and reducing the overheads associated with loops.
- It plays an important role in improving cache performance and making effective use of parallel processing capabilities.
- Most execution time of a scientific program is spent on loops; as such, many compiler optimization techniques have been developed to make them faster.

3. Loop Optimization

- Induction Variable & Reduction In Strength
- Loop Unrolling
- Loop Fission & Fusion
- Loop Invariant Code Motion
- Loop Interchange

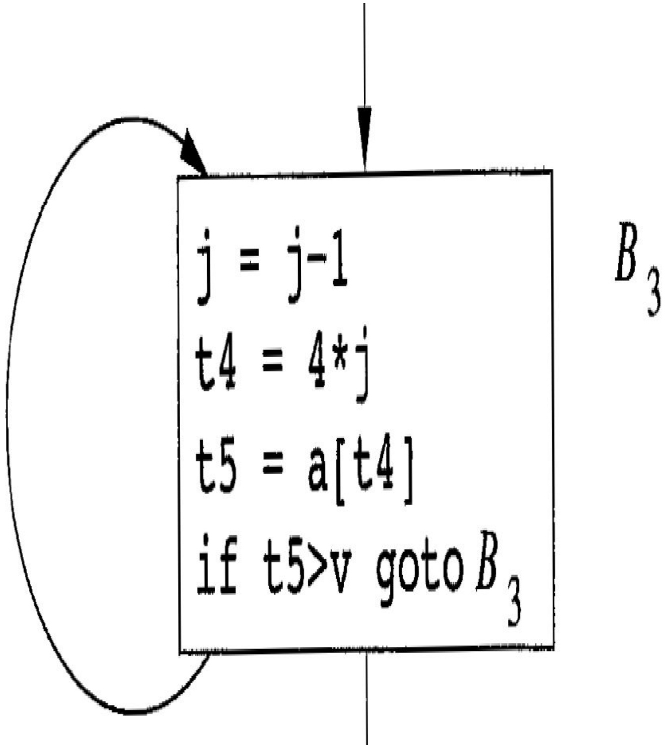
3.1 Induction Variables & Reduction in Strength

- Another important optimization is to find induction variables in loops and optimize their computation.
- A variable x is said to be an "induction variable“ if there is a positive or negative constant **c** such that each time x is assigned, its value increases by c.

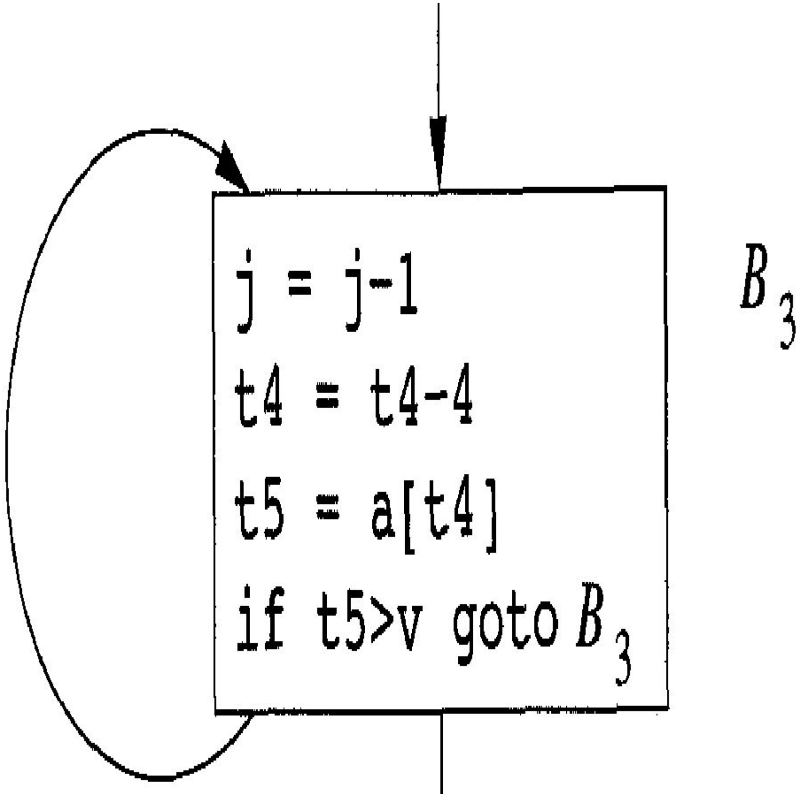
3.1 Induction Variables & Reduction in Strength

- Induction variables can be computed with a single increment (addition or subtraction) per loop iteration.
- The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as *strength reduction.*
- Induction variables allow us to perform a strength reduction.

3.1 Induction Variables & Reduction in Strength



```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```



```
j = j-1
t4 = t4-4
t5 = a[t4]
if t5>v goto B3
```

When processing loops, it is useful to work "inside-out" ; that is, we shall start with the inner loops and proceed to progressively larger, surrounding loops.

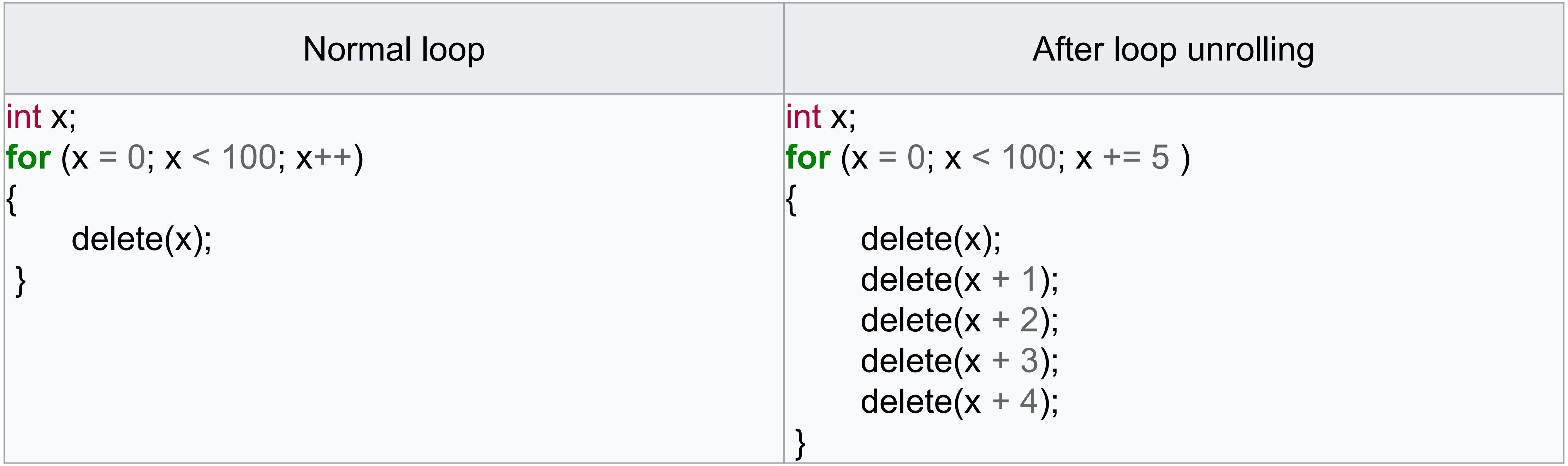
3.2 Loop Unrolling

- **Loop unrolling**, also known as **loop unwinding**, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size, which is an approach known as space–time tradeoff.
- The transformation can be undertaken manually by the programmer or by an optimizing compiler.

3.2 Loop Unrolling

- The goal of loop unwinding is to increase a program's speed by reducing or eliminating instructions that control the loop.
- Such as pointer arithmetic and "end of loop" tests on each iteration; reducing branch penalties; as well as hiding latencies including the delay in reading data from memory.
- To eliminate this computational overhead, loops can be re-written as a repeated sequence of similar independent statements.

3.2 Loop Unrolling



Normal loop	After loop unrolling

int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
	int x;
	for (x = 0; x < 100; x += 5)
	{
	    delete(x);
	    delete(x + 1);
	    delete(x + 2);
	    delete(x + 3);
	    delete(x + 4);
	}

- As a result of this modification, the new program has to make only 20 iterations, instead of 100.
- Afterwards, only 20% of the jumps and conditional branches need to be taken, and represents, over many iterations, a potentially significant decrease in the loop administration overhead.

3.3 Loop fission and fusion

- **loop fission** (or **loop distribution**) is a compiler optimization in which a loop is broken into multiple loops over the same index range with each taking only a part of the original loop's body.
- The goal is to break down a large loop body into smaller ones to achieve better utilization of locality of reference.
- This optimization is most efficient in multi-core processors that can split a task into multiple tasks for each processor.

3.3 Loop fission and fusion

Loop fission Example

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
a[i] = 1; b[i] = 2;
}
```

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
a[i] = 1;
}
for (i = 0; i < 100; i++)
{
b[i] = 2;
}
```

3.3 Loop fission and fusion

- Conversely, **loop fusion** (or **loop jamming**) is a compiler optimization and loop transformation which replaces multiple loops with a single one.
- It is possible when two loops iterate over the same range and do not reference each other's data.
- Loop fusion does not always improve run-time speed.
- On some architectures, two loops may actually perform better than one loop because, for example, there is increased data locality within each loop.

3.4 Loop-invariant code motion

- **loop-invariant code** consists of statements or expressions (which can be moved outside the body of a loop without affecting the semantics of the program.
- **Loop-invariant code motion** (also called **hoisting** or **scalar promotion**) is a compiler optimization which performs this movement automatically.

3.4 Loop-invariant code motion

- If we consider the following code sample, two optimizations can be easily applied.

```
for (int i = 0; i < n; i++)
{
x = y + z;
a[i] = 6 * i + x * x;
}
```

The calculation `x = y + z` and `x * x` can be moved outside the loop since within they are loop-invariant code, so the optimized code will be something like this:

```
x = y + z;
t1 = x * x;
for (int i = 0; i < n; i++) {
a[i] = 6 * i + t1;
}
```

3.5 Loop interchange

- **loop interchange** is the process of exchanging the order of two iteration variables used by a nested loop.
- The variable used in the inner loop switches to the outer loop, and vice versa.
- It is often done to ensure that the elements of a multi-dimensional array are accessed in the order in which they are present in memory, improving locality of reference.

3.5 Loop interchange

- For example, in the code fragment:

```
for i from 0 to 10
   for j from 0 to 20
a[i,j] = i + j
```

- loop interchange would result in:

```
for j from 0 to 20
   for i from 0 to 10
a[i,j] = i + j
```

3.5 Loop interchange

- The utility of loop interchange
  - The major purpose of loop interchange is to take advantage of the CPU cache when accessing array elements.
  - When a processor accesses an array element for the first time, it will retrieve an entire block of data from memory to cache.
  - That block is likely to have many more consecutive elements after the first one, so on the next array element access, it will be brought directly from cache (which is faster than getting it from slow main memory).
  - Cache misses occur if the contiguously accessed array elements within the loop come from a different cache block, and loop interchange can help prevent this.

3.5 Loop interchange

- The effectiveness of loop interchange depends on and must be considered in light of the cache model used by the underlying hardware and the array model used by the compiler.
- In C programming language, array elements in the same row are stored consecutively in memory (a[1,1], a[1,2], a[1,3]) – in **row-major order**.
- On the other hand, FORTRAN programs store array elements from the same column together (a[1,1], a[2,1], a[3,1]), using **column-major**.
- Optimizing compilers can detect the improper ordering by programmers and interchange the order to achieve better cache performance.

UNIT V: CODE OPTIMIZATION

Topics we are going to cover

1. Introduction– Principal Sources of Optimization
2. Optimization of basic Blocks
3. Loop Optimization
4. Runtime Environments – Source Language issues
5. Introduction to Global Data Flow Analysis
6. Storage Organization
7. Storage Allocation strategies – Access to non-local names
8. Parameter Passing.

4. Runtime Environments

- A compiler must accurately implement the abstractions embodied in the source language definition.
- These abstractions typically include the concepts such as names, scopes, bindings, data types, operators, procedures, parameters, and flow-of-control constructs.
- The compiler must cooperate with the operating system and other systems software to support these abstractions on the target machine.

4. Runtime Environments

- To do so, the compiler creates and manages a run-time environment in which it assumes its target programs are being executed.
- This environment deals with a variety of issues such as the
  - layout and allocation of storage locations for the objects named in the source program,
  - the mechanisms used by the target program to access variables,
  - the linkages between procedures,
  - the mechanisms for passing parameters,
  - and the interfaces to the operating system, input/output devices, and other programs.

UNIT V: CODE OPTIMIZATION

Topics we are going to cover

1. Introduction– Principal Sources of Optimization
2. Optimization of basic Blocks
3. Loop Optimization
4. Runtime Environments – Source Language issues
5. Introduction to Global Data Flow Analysis
6. Storage Organization
7. Storage Allocation strategies – Access to non-local names
8. Parameter Passing.

6. Storage Organization

- From the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system, and target machine.
- The operating system maps the logical addresses into physical addresses, which are usually spread throughout memory.

6 Storage Organization

- The run-time representation of an object program in the logical address space consists of data and program areas as shown in Fig.
- A compiler for a language like C++ on an operating system like Linux might subdivide memory in this way.

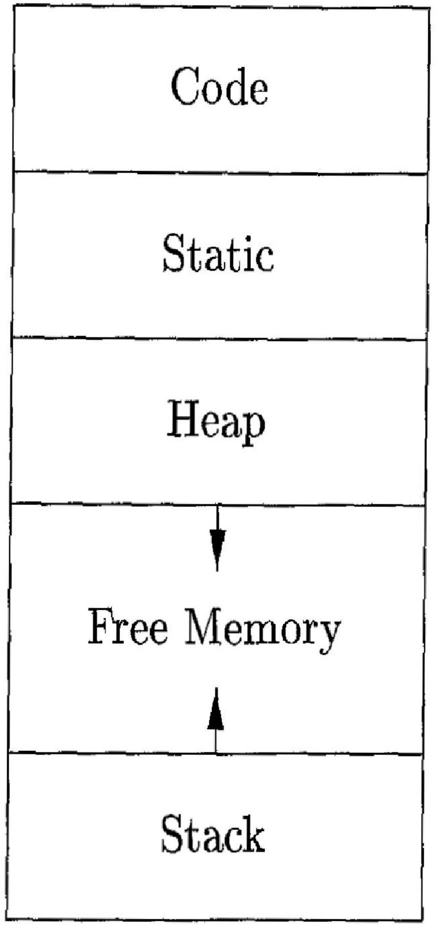
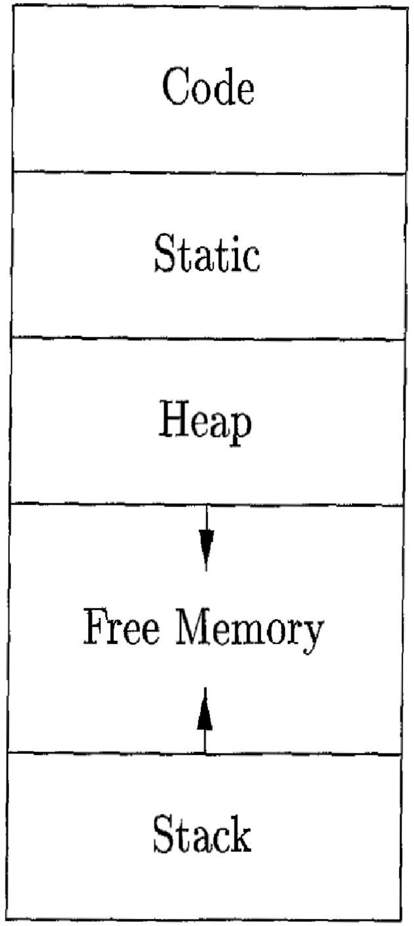


Figure: Typical subdivision of run-time memory into code and data areas

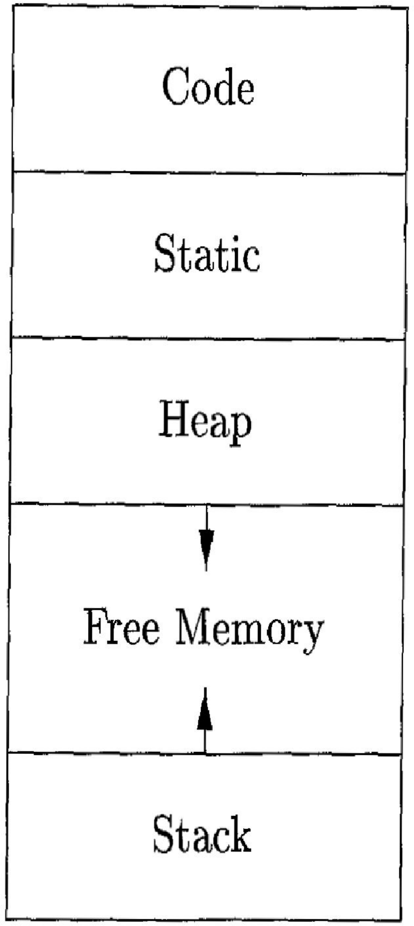
6. Storage Organization

- The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area Code, usually in the low end of memory.
- Similarly, the size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area called Static.
- One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code.



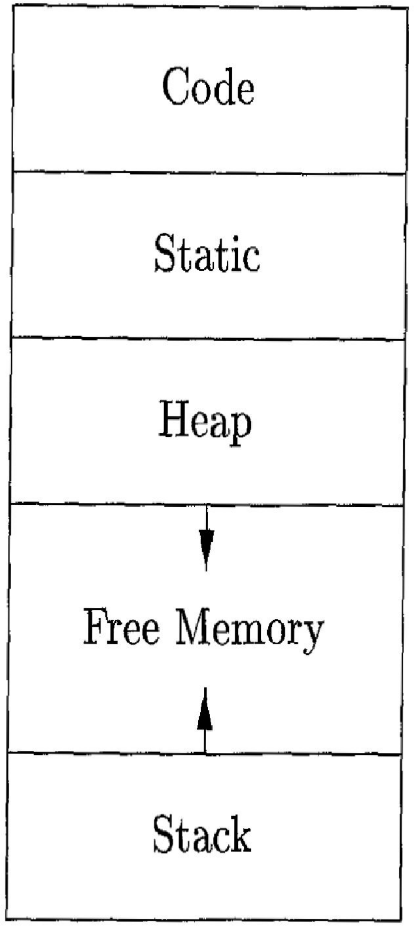
6 Storage Organization

- To maximize the utilization of space at run time, the other two areas, Stack and Heap, are at the opposite ends of the remainder of the address space.
- These areas are dynamic; their size can change as the program executes.
- These areas grow towards each other as needed. The stack is used to store data structures called activation records that get generated during procedure calls.



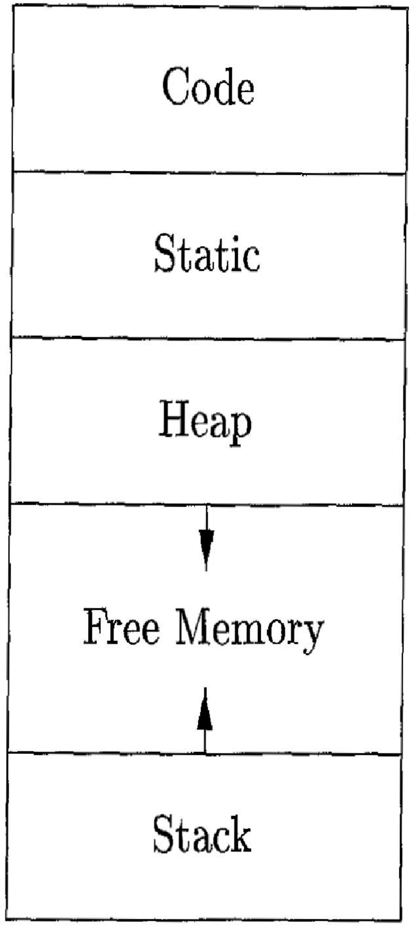
6 Storage Organization

- An activation record is used to store information about the status of the machine, such as the value of the program counter and machine registers, when a procedure call occurs.
- When control returns from the call, the activation of the calling procedure can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call.
- Data objects whose lifetimes are contained in that of an activation can be allocated on the stack along with other information associated with the activation.



6 Storage Organization

- Many programming languages allow the programmer to allocate and deallocate data under program control.
- For example, C has the functions **malloc** and **free** that can be used to obtain and give back arbitrary chunks of storage.
- The heap is used to manage this kind of long-lived data.



UNIT V: CODE OPTIMIZATION

Topics we are going to cover

1. Introduction– Principal Sources of Optimization
2. Optimization of basic Blocks
3. Loop Optimization
4. Runtime Environments – Source Language issues
5. Introduction to Global Data Flow Analysis
6. Storage Organization
7. Storage Allocation strategies – Access to non-local names
8. Parameter Passing.

7.1 Static Versus Dynamic Storage Allocation

- The layout and allocation of data to memory locations in the run-time environment are key issues in storage management.

- The two adjectives static and dynamic distinguish between compile time and run time, respectively.

- We say that a storage-allocation decision is static, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes.

- Conversely, a decision is dynamic if it can be decided only while the program is running.

7.1 Static Versus Dynamic Storage Allocation

- Many compilers use some combination of the following two strategies for dynamic storage allocation:

1. Stack storage. Names local to a procedure are allocated space on a stack. The stack supports the normal call/return policy for procedures.
2. Heap storage. Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage. The heap is an area of virtual memory that allows objects or other data elements to obtain storage when they are created and to return that storage when they are invalidated.

- To support heap management, "garbage collection" enables the run-time system to detect useless data elements and reuse their storage, even if the programmer does not return their space explicitly

7.2 Stack Allocation of Space

- Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.
- This arrangement not only allows space to be shared by procedure calls whose durations do not overlap in time, but it allows us to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same, regardless of the sequence of procedure calls.

7.2.1 Activation Tree

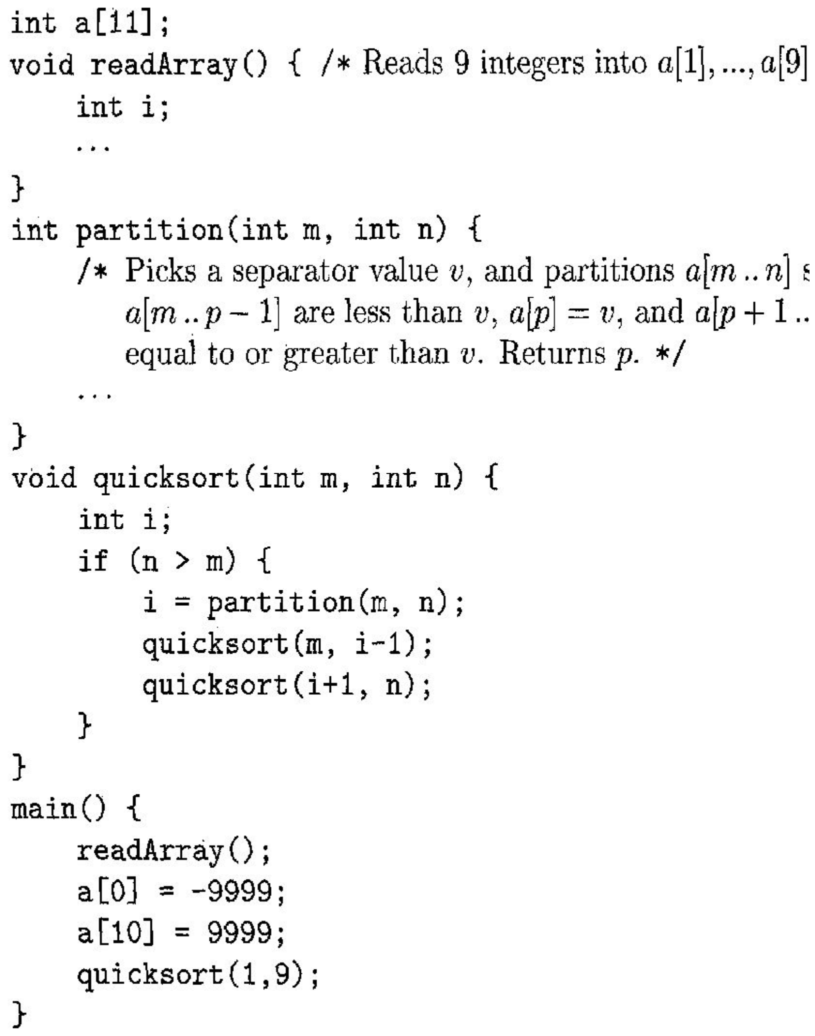
- Stack allocation would not be feasible if procedure calls, or activations of procedures, did not nest in time.
- We can represent the activations of procedures during the running of an entire program by a tree, called an *activation tree.*
- Each node corresponds to one activation, and the root is the activation of the "main" procedure that initiates execution of the program.
- At a node for an activation of procedure p, the children correspond to activations of the procedures called by this activation of p.
- We show these activations in the order that they are called, from left to right.
- Notice that one child must finish before the activation to its right can begin.

7.2.1 Activation Tree

The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:

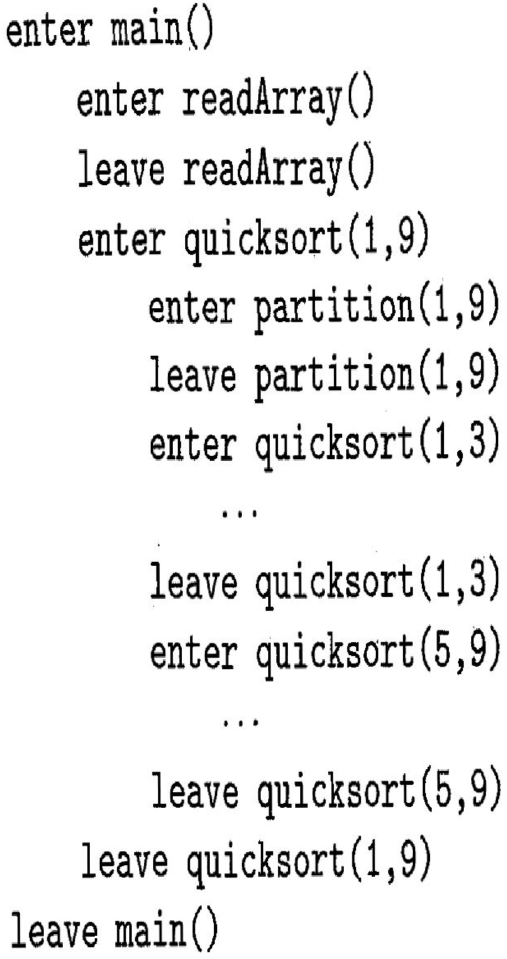
1. The sequence of procedure calls corresponds to a preorder traversal of the activation tree.
2. The sequence of returns corresponds to a postorder traversal of the activation tree.
3. Suppose that control lies within a particular activation of some procedure, corresponding to a node N of the activation tree. Then the activations that are currently open (live) are those that correspond to node N and its ancestors. The order in which these activations were called is the order in which they appear along the path to N, starting at the root, and they will return in the reverse of that order.

7.2.1 Activation Tree



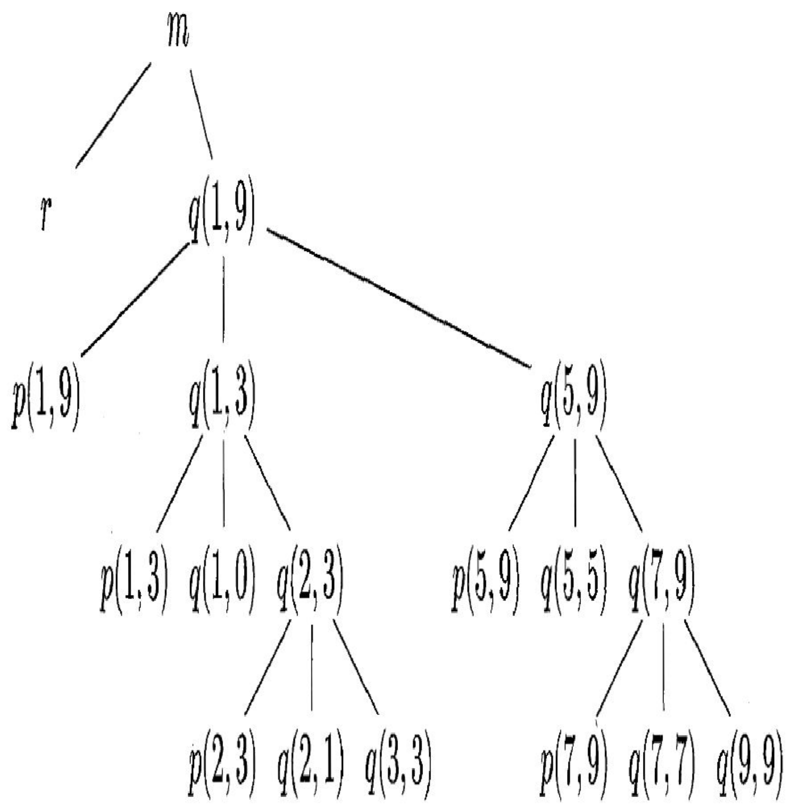
```
int a[11];
void readArray() { /* Reads 9 integers into a[1],...,a[9]
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] s
       a[m..p-1] are less than v, a[p] = v, and a[p+1..
       equal to or greater than v. Returns p. */
    ...
}
void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

Sketch of a quicksort program



```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Possible activations for the program



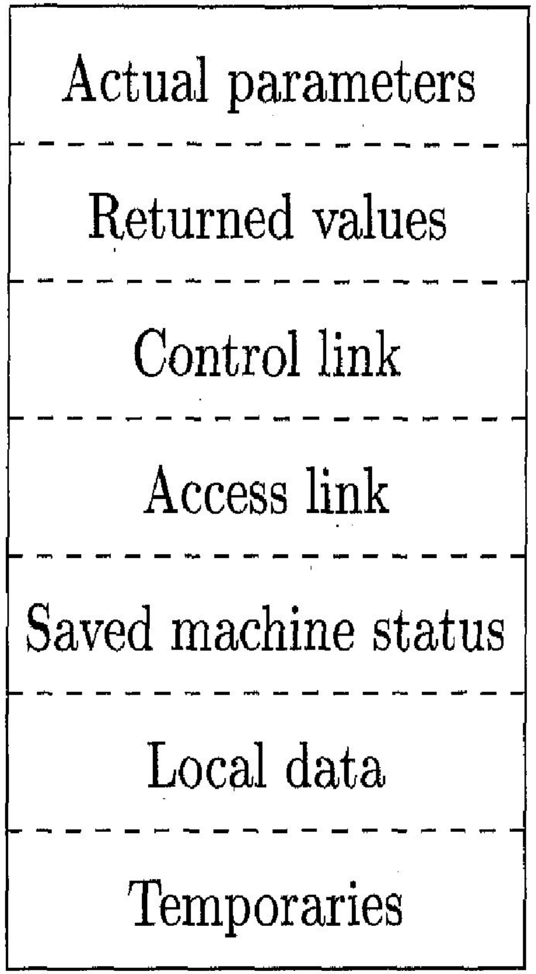
Activation tree representing calls during an execution of quicksort

7.2.2 Activation Records

- Procedure calls and returns are usually managed by a run-time stack called the control stack.
- Each live activation has an activation record (sometimes called a frame) on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.
- The latter activation has its record at the top of the stack.

7.2.2 Activation Records

- The contents of activation records vary with the language being implemented.
- Here is a list of the kinds of data that might appear in an activation record



Actual parameters

Returned values

Control link

Access link

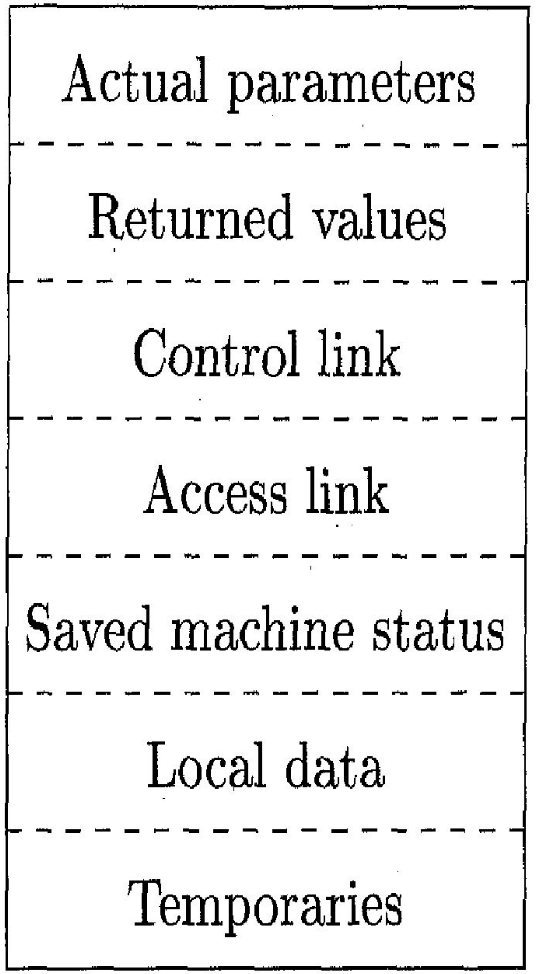
Saved machine status

Local data

Temporaries

7.2.2 Activation Records

1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.
2. Local data belonging to the procedure whose activation record this is.
3. A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.
4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.



Actual parameters

Returned values

Control link

Access link

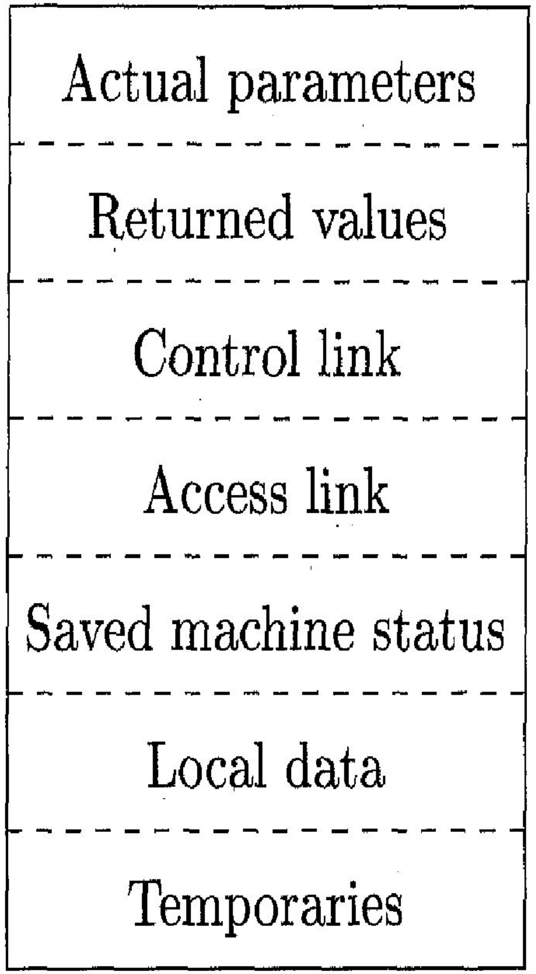
Saved machine status

Local data

Temporaries

7.2.2 Activation Records

5. A control link, pointing to the activation record of the caller.
6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
7. The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.



Actual parameters

Returned values

Control link

Access link

Saved machine status

Local data

Temporaries

7.2.3 Calling Sequences

- Procedure calls are implemented by what are known as calling sequences, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.

7.3 Heap Management

- The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it.
- While local variables typically become inaccessible when their procedures end, many languages enable us to create objects or other data whose existence is not tied to the procedure activation that creates them.
- For example, both C++ and Java give the programmer ***new*** to create objects that may be passed - or pointers to them may be passed - from procedure to procedure, so they continue to exist long after the procedure that created them is gone.
- Such objects are stored on a heap.

7.3.1 The Memory Manager

- memory manager is the subsystem that allocates and deallocates space within the heap; it serves as an interface between application programs and the operating system.
- The memory manager keeps track of all the free space in heap storage at all times.

7.3.1 The Memory Manager

- It performs two basic functions:
- *Allocation.*
  - When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size.
  - If possible, it satisfies an allocation request using free space in the heap; if no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system.
  - If space is exhausted, the memory manager passes that information back to the application program.
- *Deallocation.*
  - The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests.
  - Memory managers typically do not return memory to the operating system, even if the program's heap usage drops.

7.3.1 The Memory Manager

- Memory management would be simpler if
  - (a) all allocation requests were for chunks of the same size, and
  - (b) storage were released predictably, say, first-allocated first-deallocated.
- There are some languages, such as Lisp, for which condition (a) holds; pure Lisp uses only one data element - a two pointer cell - from which all data structures are built.
- Condition (b) also holds in some situations, the most common being data that can be allocated on the run-time stack.
- However, in most languages, neither (a) nor (b) holds in general.
- Rather, data elements of different sizes are allocated, and there is no good way to predict the lifetimes of all allocated objects.
- Thus, the memory manager must be prepared to service, in any order, allocation and deallocation requests of any size, ranging from one byte to as large as the program's entire address space.

7.3.1 The Memory Manager

Here are the properties we desire of memory managers:

- *Space Efficiency.*
  - A memory manager should minimize the total heap space needed by a program.
  - Doing so allows larger programs to run in a fixed virtual address space.
  - Space efficiency is achieved by minimizing "fragmentation.“
- *Program Efficiency.*
  - *A* memory manager should make good use of the memory subsystem to allow programs to run faster.
  - The time taken to execute an instruction can vary widely depending on where objects are placed in memory.
  - Fortunately, programs tend to exhibit "locality," a phenomenon which refers to the nonrandom clustered way in which typical programs access memory.
  - By attention to the placement of objects in memory, the memory manager can make better use of space and, hopefully, make the program run faster

7.3.1 The Memory Manager

- *Low Overhead.*
  - Because memory allocations and deallocations are frequent operations in many programs, it is important that these operations be as efficient as possible.
  - That is, we wish to minimize the *overhead* - the fraction of execution time spent performing allocation and deallocation.

UNIT V: CODE OPTIMIZATION

Topics we are going to cover

1. Introduction– Principal Sources of Optimization
2. Optimization of basic Blocks
3. Loop Optimization
4. Runtime Environments – Source Language issues
5. Introduction to Global Data Flow Analysis
6. Storage Organization
7. Storage Allocation strategies – Access to non-local names
8. Parameter Passing.

8. Parameter Passing Mechanisms

- All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments.
- In this section, we shall consider how the *actual parameters* (the parameters used in the call of a procedure) are associated with the *formal parameters* (those used in the procedure definition).

8.1 Call By Value

- In call-by-value, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable).
- The value is placed in the location belonging to the corresponding formal parameter of the called procedure.
- This method is used in C and Java, and is a common option in C++, as well as in most other languages.
- Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure, and the actual parameters themselves cannot be changed.

8.2 Call- by-Reference

- In call- by-reference, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter.
- Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller.
- Changes to the formal parameter thus appear as changes to the actual parameter.

8.2 Call- by-Reference

- If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own.
- Changes to the formal parameter change this location, but can have no effect on the data of the caller.

8.3 Call By Name

- A third mechanism - call-by-name - was used in the early programming language Algol 60.
- It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct).
- When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.