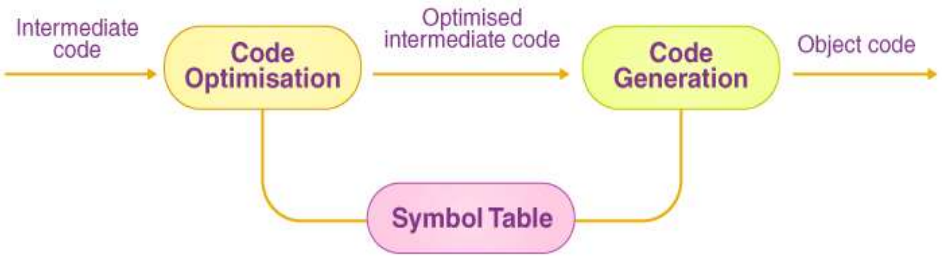
Unit 4

Code Optimization

Code optimization is a program modification strategy that enhances the intermediate code so that a program utilises the least potential memory, minimises CPU time, and offers high speed.



Reasons for Optimizing the Code

- Code optimization is essential to enhance the execution and efficiency of a source code.
- It is mandatory to deliver efficient target code by lowering the number of instructions in a program.

Role of Code Optimization

- It is the fifth stage of a compiler, and compiler will choose whether or not to optimize the code
- It helps in reducing the storage space and increases compilation speed.
- It takes source code as input and attempts to produce optimal code.
- Functioning the optimization is tedious; employing a code optimizer to accomplish the assignment is preferable.

Different Types of Optimizations

Optimization is classified broadly into two types:

1. Machine-Independent
2. Machine-Dependent

Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations.

For example:

```
do
{
  item = 10;
  value = value + item;
} while(value<100);
```

This code involves the repeated assignment of the identifier item, which, if we put it this way:

```
item = 10;
do
{
  value = value + item;
} while(value<100);
```

Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts into taking maximum advantage of the memory hierarchy.

1. Compile Time Evaluation-

Two techniques that fall under compile-time evaluation are-

A) Constant Folding-

In this technique,

As the name suggests, it involves folding the constants.

The expressions containing the operands with constant values at compile time are evaluated.

Those expressions are then replaced with their respective results.

Example-

Circumference of Circle = (22/7) x Diameter

Here,

This technique evaluates the expression 22/7 at compile time.

The expression is then replaced with its result 3.14.

This saves the time at run time.

B) Constant Propagation-

In this technique,

If some variable has been assigned some constant value, it replaces that variable with its constant value in the further program during compilation.

The condition is that the variable's value must not be altered in between.

Example-

pi = 3.14

radius = 10

Area of circle = pi x radius x radius

Here,

This technique substitutes the value of variables ‘pi’ and ‘radius’ at compile time.

It then evaluates the expression 3.14 x 10 x 10.

The expression is then replaced with its result 314.

This saves the time at run time.

2. Common Sub-Expression Elimination-

The expression already computed before and appears again in the code for computation is called Common Sub-Expression.

In this technique,

- As the name suggests, it involves eliminating the common sub-expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

Code Before Optimization	Code After Optimization
S1 = 4 x i S2 = a[S1] S3 = 4 x j S4 = 4 x i **// Redundant Expression** S5 = n S6 = b[S4] + S5	S1 = 4 x i S2 = a[S1] S3 = 4 x j S5 = n S6 = b[S1] + S5

3. Code Movement-

In this technique,

- As the name suggests, it involves the movement of the code.
- The code inside the loop is moved out if it does not matter whether it is inside or outside.
- Such a code unnecessarily gets executed repeatedly with each iteration of the loop.
- This leads to the wastage of time at run time.

Code Before Optimization	Code After Optimization

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

4. Dead Code Elimination-

In this technique,

- As the name suggests, it involves eliminating the dead code.
- The statements of the code that either never execute or are unreachable or whose output is never used are eliminated.

Code Before Optimization	Code After Optimization

i = 0 ;
if (i == 1)
{
a = x + 5 ;
}
	i = 0 ;

5. Strength Reduction-

In this technique,

- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and inexpensive ones.

Code Before Optimization	Code After Optimization
B = A x 2	B = A + A

Here,

The expression “A x 2” is replaced with the expression “A + A”.

This is because the cost of the multiplication operator is higher than that of the addition operator.