

Memory placement strategies



Write a Java/C++ program to simulate memory placement strategies 1. First Fit 2. Best Fit 3. Worst Fit 4. Next Fit

```
#include <iostream>
using namespace std;

int blocks[] = {100, 500, 200, 300, 600};
int processes[] = {212, 417, 112, 426};
int blkSize = sizeof(blocks) / sizeof(blocks[0]);
int prSize = sizeof(processes) / sizeof(processes[0]);

void FirstFit() {
    cout<<endl<<"FIRST FIT:";
    for (int i=0; i<prSize; i++) {
        bool check = false;
        for(int j=0; j<blkSize; j++) {
            if ((blocks[j] - processes[i] >= 0)) {
                blocks[j] -= processes[i];
                check = true;
                cout<<endl<<"Process "<<processes[i]<<" has been allocated to block "<<j+1;
                break;
            }
        }
        if (check == false) {
            cout<<endl<<"Process "<<processes[i]<<" has not been allocated.";
        }
    }
}

void NextFit() {
    int j = 0;
    cout<<endl<<"NEXT FIT:";
    for (int i=0; i<prSize; i++) {
        bool check = false;
        for (int k=0; k<blkSize; k++) {
            if ((blocks[j] - processes[i] >= 0) {
                blocks[j] = blocks[j] - processes[i];
                cout<<endl<<"Process "<<processes[i]<<" has been allocated to block "<<j+1;
                check = true;
                break;
            }
        }
        j = (j + 1) % blkSize;
    }
    if (check == false) {
        cout<<endl<<"Process "<<processes[i]<<" has not been allocated.";
    }
}
```

```

    }
}

void BestFit() {
    cout<<endl<<"BEST FIT:";
    for (int i=0; i<prSize; i++) {
        bool check = false;
        int k, store = 999;
        for (int j=0; j<blkSize; j++) {
            if ((blocks[j] - processes[i]) >= 0 && (blocks[j] - processes[i]) < store) {
                k = j + 1;
                store = blocks[j] - processes[i];
                check = true;
            }
        }
        if (check == false) {
            cout<<endl<<"Process "<<processes[i]<<" was not allocated to any block.";
        }
        else {
            blocks[k - 1] -= processes[i];
            cout<<endl<<"Process "<<processes[i]<<" was allocated to block "<<k;
        }
    }
}

void WorstFit() {
    cout<<endl<<"WORST FIT:";
    for (int i=0; i<prSize; i++) {
        bool check = false;
        int k, store = -1;
        for (int j=0; j<blkSize; j++) {
            if ((blocks[j] - processes[i]) >= 0 && (blocks[j] - processes[i] > store)) {
                k = j + 1;
                store = blocks[j] - processes[i];
                check = true;
            }
        }
        if (check == false) {
            cout<<endl<<"Process "<<processes[i]<<" was not allocated to any block.";
        }
        else {
            blocks[k - 1] -= processes[i];
            cout<<endl<<"Process "<<processes[i]<<" was allocated to block "<<k;
        }
    }
}

int main () {
    // ONLY RUN ONE FUNCTION AT A TIME.
    // FirstFit();
}

```

```
// NextFit();
BestFit();
// WorstFit();
return 0;
}
```

explain code as well as give input for running it

Let's go through your **Memory Placement Strategies** program in detail — explanation, working, and example input/output — so you can understand it fully and run it easily in Ubuntu or any C++ environment.

Concept Overview

When a process requests memory, the operating system can allocate it using **different placement (fit) strategies**:

Strategy	Description
First Fit	Allocates the <i>first</i> memory block that is large enough.
Best Fit	Allocates the <i>smallest block</i> that can fit the process (minimizes wasted space).
Worst Fit	Allocates the <i>largest block</i> that can fit the process (maximizes leftover space).
Next Fit	Similar to First Fit, but continues searching from the <i>last allocated position</i> instead of starting at the beginning.

Code Explanation (Step by Step)

Global Data

```
cpp

int blocks[] = {100, 500, 200, 300, 600};
int processes[] = {212, 417, 112, 426};
```

- **blocks[]** → memory partitions (in KB or MB).
- **processes[]** → processes with their memory requirements.

```
cpp

int blkSize = sizeof(blocks) / sizeof(blocks[0]);
int prSize = sizeof(processes) / sizeof(processes[0]);
```

- Calculates number of blocks (**blkSize = 5**) and processes (**prSize = 4**).

1. First Fit

cpp

```
void FirstFit() {
    for (int i = 0; i < prSize; i++) {
        bool check = false;
        for (int j = 0; j < blkSize; j++) {
            if ((blocks[j] - processes[i]) >= 0) {
                blocks[j] -= processes[i];
                check = true;
                cout << "\nProcess " << processes[i] << " allocated to block " << j + 1;
                break;
            }
        }
        if (!check) cout << "\nProcess " << processes[i] << " not allocated.";
    }
}
```

✓ Logic:

- For each process, look from the *first block onward*.
- Allocate the first block that can fit it.
- Reduce block size (since part of it is now used).

⚙ 2. Next Fit

cpp

```
void NextFit() {
    int j = 0;
    for (int i = 0; i < prSize; i++) {
        bool check = false;
        for (int k = 0; k < blkSize; k++) {
            if ((blocks[j] - processes[i]) >= 0) {
                blocks[j] -= processes[i];
                cout << "\nProcess " << processes[i] << " allocated to block " << j + 1;
                check = true;
                break;
            }
            j = (j + 1) % blkSize; // wrap-around
        }
        if (!check) cout << "\nProcess " << processes[i] << " not allocated.";
    }
}
```

✓ Logic:

- Like First Fit, but it **starts searching from the last allocated position**.
- Uses circular search (% `blkSize`) so it wraps around.

⚙ 3. Best Fit

cpp

```
void BestFit() {
    for (int i = 0; i < prSize; i++) {
        bool check = false;
        int k, store = 999;
        for (int j = 0; j < blkSize; j++) {
            if ((blocks[j] - processes[i]) >= 0 && (blocks[j] - processes[i]) < store) {
```

```

        k = j + 1;
        store = blocks[j] - processes[i];
        check = true;
    }
}
if (!check) cout << "\nProcess " << processes[i] << " not allocated.";
else {
    blocks[k - 1] -= processes[i];
    cout << "\nProcess " << processes[i] << " allocated to block " << k;
}
}
}

```

✓ Logic:

- For each process, find the block that will leave the **least leftover space**.
- Minimizes fragmentation.

⚙️ 4. Worst Fit

cpp

```

void WorstFit() {
    for (int i = 0; i < prSize; i++) {
        bool check = false;
        int k, store = -1;
        for (int j = 0; j < blkSize; j++) {
            if ((blocks[j] - processes[i]) >= 0 && (blocks[j] - processes[i]) > store) {
                k = j + 1;
                store = blocks[j] - processes[i];
                check = true;
            }
        }
        if (!check) cout << "\nProcess " << processes[i] << " not allocated.";
        else {
            blocks[k - 1] -= processes[i];
            cout << "\nProcess " << processes[i] << " allocated to block " << k;
        }
    }
}

```

✓ Logic:

- Opposite of Best Fit — find the **largest possible block** for each process.
- Tries to avoid too many small fragments.

⌚ Main Function

cpp

```

int main() {
    // Run only one strategy at a time
    // FirstFit();
    // NextFit();
    BestFit();
    // WorstFit();
    return 0;
}

```

You can uncomment the algorithm you want to run.

💻 How to Run in Ubuntu Terminal

Step 1 — Save file

Create file `memory_fit.cpp`:

```
bash  
nano memory_fit.cpp
```

Paste the entire code, save (**Ctrl+O, Enter**), then exit (**Ctrl+X**).

Step 2 — Compile

```
bash  
g++ memory_fit.cpp -o memory_fit
```

Step 3 — Run

```
bash  
../memory_fit
```

⬇️ Sample Input

(Already hardcoded in the program)

```
cpp  
  
blocks[] = {100, 500, 200, 300, 600};  
processes[] = {212, 417, 112, 426};
```

⬆️ Sample Output

♦ For Best Fit

```
arduino  
  
BEST FIT:  
Process 212 allocated to block 4  
Process 417 allocated to block 2  
Process 112 allocated to block 3  
Process 426 not allocated.
```

♦ For First Fit

```
arduino
```

FIRST FIT:
 Process 212 allocated to block 2
 Process 417 allocated to block 5
 Process 112 allocated to block 3
 Process 426 not allocated.

◆ For Worst Fit

```
arduino
```

WORST FIT:
 Process 212 allocated to block 5
 Process 417 allocated to block 2
 Process 112 allocated to block 4
 Process 426 not allocated.

◆ For Next Fit

```
arduino
```

NEXT FIT:
 Process 212 allocated to block 2
 Process 417 allocated to block 5
 Process 112 allocated to block 1
 Process 426 not allocated.

Summary Table

Strategy	Allocation Order	Pros	Cons
First Fit	First available block	Simple, fast	Can cause external fragmentation
Next Fit	Starts after last allocation	Avoids re-scanning	Still fragments
Best Fit	Smallest possible block	Efficient space usage	Slow, may create small unusable gaps
Worst Fit	Largest block	Reduces fragmentation early	May waste space

Would you like me to modify this code so that **user inputs block and process sizes dynamically** instead of hardcoding them?



now give viva 25 question along with solution for the same code

Excellent — here are **25 important viva (oral exam) questions and answers** based on your **Memory Placement Strategies (First Fit, Best Fit, Worst Fit, Next Fit)** C++ simulation program.

These are exactly the kind of questions your examiner may ask — including conceptual, code-based, and analytical ones.

Viva Questions and Answers (Memory Placement Strategies)

1. What is memory allocation in an operating system?

Answer:

Memory allocation is the process by which the operating system assigns portions of main memory to programs and data while they are executing.

2. What is the difference between fixed and variable partition memory allocation?

Answer:

- **Fixed Partition:** Memory is divided into fixed-size blocks; each process fits into one block.
 - **Variable Partition:** Memory is divided dynamically based on process size at runtime.
-

3. What is the main purpose of memory placement strategies?

Answer:

They decide **where in memory** a new process should be placed when multiple memory blocks are available — aiming to reduce fragmentation and optimize memory use.

4. Explain the First Fit memory allocation strategy.

Answer:

First Fit allocates the first block that is large enough to accommodate the process. It searches from the beginning of the memory list each time.

5. What is the main advantage of First Fit?

Answer:

It is **simple and fast**, as it stops searching after finding the first suitable block.

6. What is the disadvantage of First Fit?

Answer:

It causes **external fragmentation** and tends to fill up lower memory areas quickly, leaving small unusable holes.

7. Explain the Best Fit allocation strategy.

Answer:

Best Fit allocates the smallest available block that can hold the process, aiming to minimize leftover space (internal fragmentation).

8. What is the drawback of Best Fit?

Answer:

It can create many **small unusable gaps** (external fragmentation) and requires scanning all blocks, making it slower.

9. Explain the Worst Fit allocation strategy.

Answer:

Worst Fit allocates the **largest available block**, leaving the biggest leftover space to reduce fragmentation over time.

10. What is the main disadvantage of Worst Fit?

Answer:

It can lead to **inefficient memory use** because large blocks are split unnecessarily, and small processes may waste big portions of memory.

11. What is the Next Fit allocation strategy?

Answer:

Next Fit works like First Fit, but instead of always starting from the beginning, it starts searching **from the last allocated position**.

12. How does Next Fit help compared to First Fit?

Answer:

It reduces allocation time for consecutive requests and distributes allocations more evenly throughout memory.

13. What is the difference between internal and external fragmentation?

Answer:

- **Internal fragmentation:** Wasted space *inside* allocated blocks.
 - **External fragmentation:** Free memory exists but is scattered in small, non-contiguous parts.
-

14. Which strategy causes the least fragmentation generally?

Answer:

Best Fit usually causes less internal fragmentation but may increase external fragmentation; there's no perfect method — it depends on workload.

15. What data structures are used in this program?

Answer:

Simple **arrays** (`blocks[]`, `processes[]`) are used to store memory block sizes and process sizes.

16. How does the program check if a process can be allocated to a block?**Answer:**

By checking if `blocks[j] - processes[i] >= 0`, meaning the block has enough free space for the process.

17. How is memory reduced after allocation in the program?**Answer:**

Once a process is allocated, the program subtracts the process size from the block size:

```
cpp  
blocks[j] -= processes[i];
```

18. What will happen if a process is larger than any available block?**Answer:**

The process will **not be allocated** and a message like

Process X has not been allocated.

will be displayed.

19. Why do we use `sizeof(blocks)/sizeof(blocks[0])` in the program?**Answer:**

It calculates the number of elements in the array, i.e., total number of memory blocks.

20. What is the output when all blocks are smaller than process size?**Answer:**

Each process will output a message saying "**Process not allocated.**", because no block can accommodate it.

21. Why should we run only one algorithm at a time in `main()`?**Answer:**

Because each function modifies the `blocks[]` array (reducing available memory). Running multiple functions consecutively gives wrong results.

22. How can we reset block sizes for multiple runs?**Answer:**

Either reinitialize `blocks[]` before each call or create separate copies for each algorithm.

23. What is the time complexity of First Fit?

Answer:

$O(n \times m)$, where

- n = number of processes
- m = number of memory blocks
(because for each process, it may need to scan all blocks).

24. What happens in the program if you use modulo % blkSize in Next Fit?

Answer:

It ensures that the search wraps around from the end of the array to the beginning (circular search).

25. How can this program be improved?

Answer:

- Take **user input** for block and process sizes.
- Display **allocation table** (showing which process goes to which block).
- Show **fragmentation values**.
- Allow **multiple runs** without reinitialization.



Bonus — Example Viva Discussion Snippet

Examiner: What would happen if a process of size 700KB is to be allocated with these blocks?

Student: Since the largest block is 600KB, the process will not be allocated in any of the strategies. The program will print "Process 700 was not allocated."

Would you like me to also add a **table output version** of this program (showing allocation result in a neat table with internal fragmentation)?

That's often asked in viva and output printouts.