AdvEmbSoft

C++

Initializes a vector `v` with four integers and uses a range-based for loop to print each element. The `auto` keyword automatically deduces the element type (`int` in this case).

```
std::vector<int> v = {7, 5, 16, 8};
for (auto n : v) {
    std::cout << n << " ";
}
```

Objects can be allocated on the stack or heap.`s1` and `s2` are stack-allocated strings, while `s3` is a heap-allocated string. `s4` is a deep copy of `s2`, and `s5` is a shallow copy of `s3`. Changes to `s5` affect `s3`, as shown by changing the first character to '@'. Heap memory is deallocated with `delete`.

```
std::string s1; //empty string
std::string s2("Hello"); //initialized string
std::string* s3 = new std::string("World"); //heap string
std::string s4 = s2; //deep copy
std::string* s5 = s3; //shallow copy
s5->at(0) = '@'; //s3 is also modified
delete s3; s3 = nullptr; //heap deallocation
delete s5; s5 = nullptr;
```

Declares two global variables. `globalVariable` can be accessed from any part of the program, while `restrictedGlobalVariable` is `static`, limiting its scope to this file only.

```
int globalVariable = 9;
static int restrictedGlobalVariable = 10;
```

Declares global constants and class constants. `kGlobalConstantExpr` is a compile-time constant, while `kGlobalConstant` is a runtime constant. Inside `ClassWithConstant`, `kConstantExpr` is a compile-time constant, and `kConstant` is a runtime constant initialized with `kGlobalConstant * 2`.

```
static constexpr uint32_t kGlobalConstantExpr = 1;
static const uint32_t kGlobalConstant = rand();
class ClassWithConstant {
public:
    static constexpr uint32_t kConstantExpr = 2;
    static const uint32_t kConstant;
};
const uint32_t ClassWithConstant::kConstant = kGlobalConstant * 2;
```

Demonstrates the use of fixed and dynamic arrays. `myArray1` is a fixed-size array on the stack, while `myArray2` is a dynamically allocated array on the heap. `myArray` is a dynamic array whose size is determined at runtime. The `sizeof` operator is used to calculate the size of `myArray`.

```
static constexpr int arraySize = 30;
int myArray1[arraySize] = {0}; //fixed array on stack
int* myArray2 = new int[arraySize];  //dynamic array on heap
delete[] myArray2; //realease heap memory
myArray2 = nullptr; //reset pointer
int varSize = 5;
int myArray[varSize] = {0};
myArray[0] = 1;
myArray[1] = myArray[0];
int size = sizeof(myArray) / sizeof(int);
```

Demonstrates the use of a `std::vector`, a dynamic array. It initializes `v` with four integers, adds two more at the end, removes the last one, and accesses the first and last elements. It also gets the size of the vector.

```
std::vector<int> v = {7, 5, 16, 8};
v.push_back(25); //add element to end
v.push_back(13); //add another element to end
v.pop_back(); //remove last element
std::cout << "First element is : " << v.front() << std::endl;
std::cout << "Last element is : " << v.back() << std::endl;
int size = v.size();
```

Two ways to create a 2D array. The first part uses raw pointers to create a jagged array, where each row has a different length. The second part uses `std::vector`, a dynamic array from the C++ Standard Library, to achieve the same result in a safer and simpler way.

```
char** array = new char*[base]; //jagged array
for (int i = 0; i < base; i++) {
    array[i] = new char[base - i + 1];
    for (int j = 0; j < base - i; j++){
        array[i][j] = '*';
    }
    array[i][base - i] = '\0';
}

std::vector<std::vector<char>> array(base); //2D vector
for (int i = 0; i < base; i++){
    array[i] = std::vector<char>(base - i, '*');
}
```

Defines a `Point` class with two constructors, methods to move the point and access its coordinates, and an operator overload for addition. The '+' operator is overloaded to add the coordinates of two points.

```
class Point {
    public:
        Point() : _x(0), _y(0) { } // constructors
        Point(float x, float y) : _x(x), _y(y) { }
        ~Point() { // destructor
            std::cout << "Point destructor called" << std::endl;
        }
        void move(float dx, float dy) { // public methods
            _x += dx;
            _y += dy;
        }
        float x() const { return _x; }
        float y() const { return _y; }
        Point operator+(const Point& other) const {
            return Point(_x + other._x, _y + other._y);
        }
    private:
        float _x; // private data fields
        float _y;
};
```

Demonstrates the concept of polymorphism. It defines a `Base` class and a `Derived` class that inherits from `Base`. Both classes have a method `f()`, but in `Derived`, `f()` is overridden. When `f()` is called on a `Base` reference or pointer that actually points to a `Derived` object, the `Derived` version of `f()` is called due to the `virtual` keyword.

```
class Base {
public:
    virtual void f() {
        std::cout << "f() in Base class called" << std::endl;
    }
};
class Derived : public Base {
public:
    void f() override {
        std::cout << "f() in Derived class called" << std::endl;
```

```cpp
    }
};
int main() {
    Base b; // Create a base instance
    Derived d; // Create a derived instance
    b.f(); // Prints base
    d.f(); // Prints derived
    Base& br = b; // The type of br is Base&
    Base& dr = d; // The type of dr is Base& as well
    br.f(); // Prints base
    dr.f(); // Prints derived (because Base::f() is declared as virtual)
    Base* bp = &b; // The type of bp is Base*
    Base* dp = &d; // The type of dp is Base* as well
    bp->f(); // Prints base
    dp->f(); // Prints derived (because Base::f() is declared as virtual)
    br.Base::f(); // prints base
    dr.Base::f(); // prints base
    return 0;
}
```

Defines a `Buffer` class that allocates a block of memory when it is constructed and releases that memory when it is destroyed. This is a common pattern for classes that manage resources, which is often referred to as Resource Acquisition Is Initialization (RAII).

```cpp
class Buffer {
public:
    Buffer(size_t size) : _size(size), _data(new char[size]) { }
    ~Buffer() { // destructor
        delete[] _data; // release the allocated memory
    }
private:
    size_t _size;
    char* _data;
};
```

Demonstrates two ways of managing dynamic memory. The first part uses raw pointers to allocate and deallocate memory manually, which can lead to memory leaks if not handled properly. The second part uses `std::unique_ptr`, a smart pointer that automatically deallocates the memory when it's no longer needed, preventing memory leaks.

```cpp
void function()
{
    char* pArray = new char[100]; // solution with raw pointer
    bool condition = false;
    if (condition) {
        // at this point, a memory leak arises since pArray is not released
        return;
    }
    delete [] pArray;
    pArray = nullptr;
    // solution with unique_ptr
    std::unique_ptr<char[]> array_ptr(new char[99]);
    if (condition) {
        // no memory leak since the destructor of array_ptr is called
        return;
    }
}
```

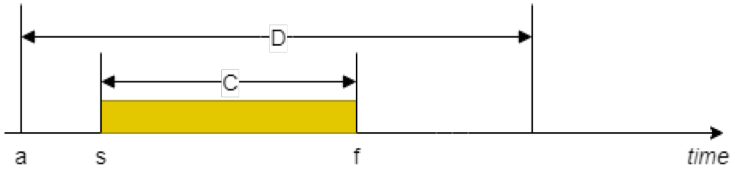
Class templates allow for creating classes that can work with different data types and sizes. The `Arithmetic<T>` class performs basic arithmetic operations on any type `T`. The `Queue<T, queue_sz>` class represents a queue of any type `T` with a size `queue_sz` known at compile time.

```cpp
template <class T> class Arithmetic {
public:
    Arithmetic(T a, T b) : _a(a), _b(b) {}
    T add() { return _a + _b; }
    T sub() { return _a - _b; }
private:
    T _a, _b;
};
template <typename T, uint32_t queue_sz> class Queue {
public:
    uint32_t getSize() const { return queue_sz;}
private:
    T array[queue_sz];
};
int main() {
    Arithmetic<float> a2(1.5, 2.1);
    Queue<int, 10> intQueue;
    return 0;
}
```

Scheduling

Timing constraints:

Deadline: D
Arrival time: A
Release time: R
Finish time: F
Computation time: C



Task response time $= f - a$ ou $f - r$

Maximum lateness $= \max_i(f_i - d_i)$

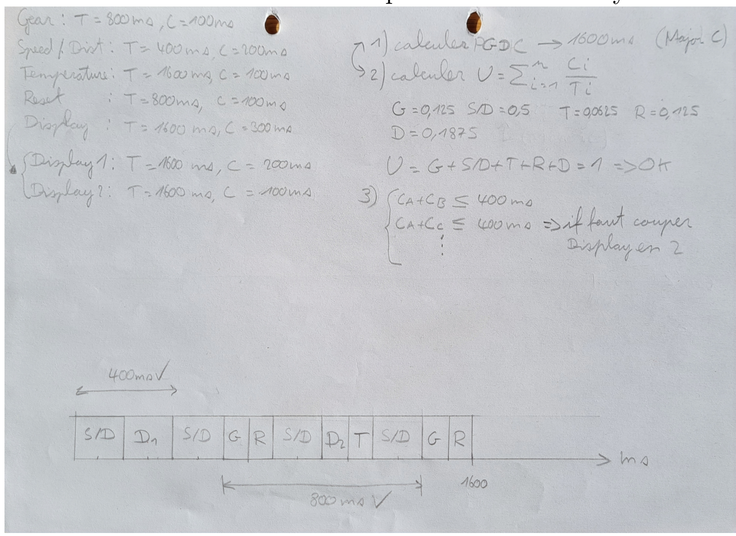
Maximum number of late tasks $= \sum_{i=1}^{n} \text{miss}(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$

Schedulability: $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$

Can be increased by increasing task computation times or by decreasing their periods.

Time cyclic scheduling:

Divide the temporal axis into slots of equal length (length = Minor Cycle). The optimal length of the Minor Cycle is the GCD (=plus grand diviseur commun) of the periods. The optimal length of the Major Cycle is the LCM (=plus petit multiple commun) of the periods. If tasks cannot be split into sub-tasks, then a set of tasks is schedulable if the sum of execution times within each time slot is less or equal to the Minor Cycle.



Advantages:

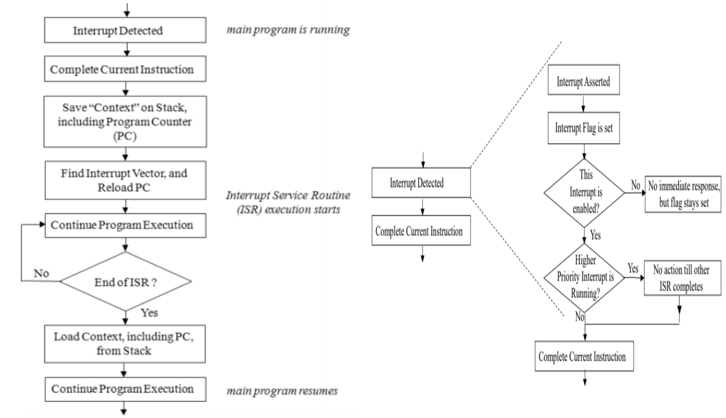
- Easy to implement
- Predictable
- No need for a scheduler

Disadvantages:

- Always run the same schedule

- The polling rate is limited by 1/maximum delay.
- The program must continually check the status of every device
- This approach scales badly

Event-driven systems:
Handling events is done using interrupts. Interrupts can be prioritized. Interrupts can be masked: switched off if not needed or likely to get in the way of more important activity. Interrupts can be nested. The location of the ISR in memory can be selected.

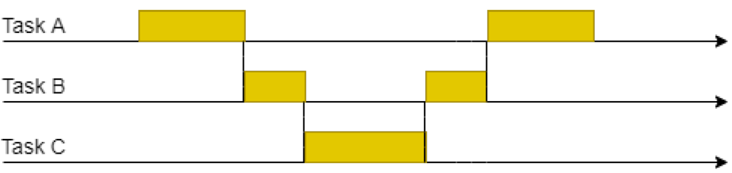


Dynamic scheduling:
Allows task scheduling to be computed dynamically online based on importance or any other criteria such as task deadline, duration, or creation time.
Task preemption:
Exception handling/interrupt may need to preempt existing tasks. Suspend a running task and insert it in the ready queue. Schedule tasks based on their priority/importance. Improve efficiency.
Advantages:

- Better response time
- Can do more processing
- Can lower processor speed, saving money and power

Disadvantages:

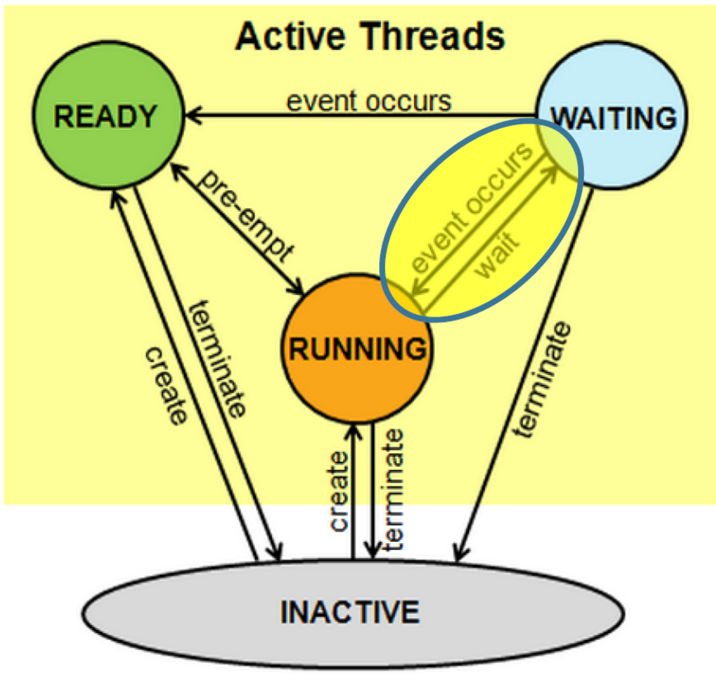
- More complicated programming
- More memory.
- Introduces vulnerability



Rate Monotonic Scheduling:
Tasks with higher request rates/shorter periods have higher priorities. Fixed periods means fixed priorities. Is optimal among fixed-priority algorithms. *Schedulability:* $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$
Earliest Deadline First:
Tasks with earlier deadlines have higher priorities. Priorities are dynamic since absolute deadlines of periodic tasks vary over time. *Schedulability:* $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$

Tasks and concurrency

A task is a set of program instructions, a process is an instance of a computer program that is being executed, and a thread is a basic unit of CPU utilization that can exist within a process. Meaning a thread consists of a program counter, a stack, and a set of registers.
Each thread has its own stack, but all threads in a process share the same heap. The stack is used for storing temporary data that is created and destroyed during the execution of a function, while the heap is used for storing data that needs to persist beyond the lifetime of a function.



In a multitasking system, the different tasks may compete for shared resources or may wait for different events to happen. In some cases, a given task may thus enter a Waiting or Blocked state.
Mutex: only one thread can access a shared resource at a time.

```
Mutex mtx;
mtx.lock();
//critical section
mtx.unlock();
```

Semaphore:
A semaphore is a synchronization object that controls access to a shared resource by multiple threads. Unlike a mutex, a semaphore can control access to several shared resources.

```
Semaphore sem_in {2};
Semaphore sem_out {0};
sem_in.acquire(); //decrement
//critical section
sem_out.release(); //increment
```

Deadlock:
A deadlock is a situation where two or more competing tasks are waiting for each other to finish, and thus neither ever does.
Queue/Mail:
A queue is a FIFO data structure that can be used to pass messages between tasks.

Priority inversion

Priority inversion occurs when a higher-priority task is waiting for a resource held by a lower-priority task. The main sources of priority inversion are non preemptable sections, sharing resources. synchronization and mutual exclusion. The solution is to use Resource Access Protocols
Non-Preemptive Protocol (NPP)
A task is assigned the highest priority if it succeeds in locking a critical section. The task is assigned its own priority when it releases the critical section.
Advantages:

- Bounds(= limite) Priority Inversion and for a given task the bound is the maximal length of any single critical section belonging to lower priority tasks.
- It is deadlock free and limits the number of blocking of any task to one.

Disadvantages:

- It allows low priority tasks to block high priority tasks.

Priority Inheritance Protocol (PIP)
The idea is to elevate the priority of a low priority task to the highest priority of tasks blocked by it. And resume its original priority when it exits the critical section. This prevents medium-priority tasks from preempting lower priority tasks and thus prolonging the blocking duration experienced by the higher-priority tasks.
Advantages:

- Blocking time is bounded.
- Blocking time can be computed.

Disadvantages:

- It is not deadlock free.
- Chain blocking can occur.

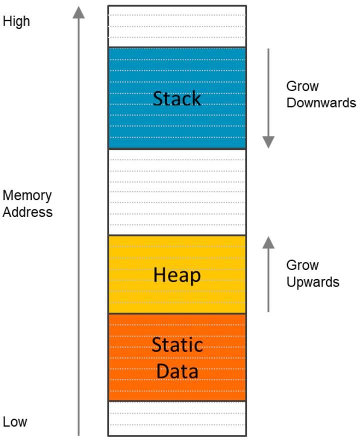
Highest locker priority protocol (HLP)

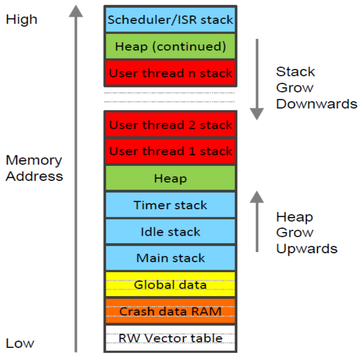
Define the ceiling $C(S)$ of a critical section S to be the highest priority of all tasks that use S during execution. Note that $C(S)$ can be calculated statically (off-line). When it finishes with S, it sets its priority back to what it was before.

Algorithms	Chain blocking	Unnecessary blocking	Blocking instant	Deadlock prevention	Transparency	Implementation
NPP	No	Yes	arrival	yes	Yes	Easy
PIP	Yes	limited	access	no	Yes	medium
HLP	No	Yes	arrival	yes	No	medium

Memory

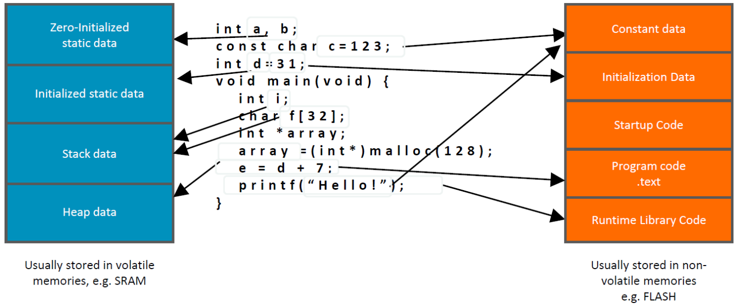
RAM/ROM:





Typically, the data can be divided into three sections: static data, stack, and heap:

- Static data: contains global variables and static variables
- Stack: contains the temporary data for local variables, parameter passing in function calls, registers saving during exceptions, etc.
- Heap: contains the pieces of memory spaces that are dynamically reserved by calloc() malloc() or new calls.



Can the information change?

- No: put it in read-only, nonvolatile memory for saving RAM
- Yes: put it in read/write memory

How long does the data need to exist?

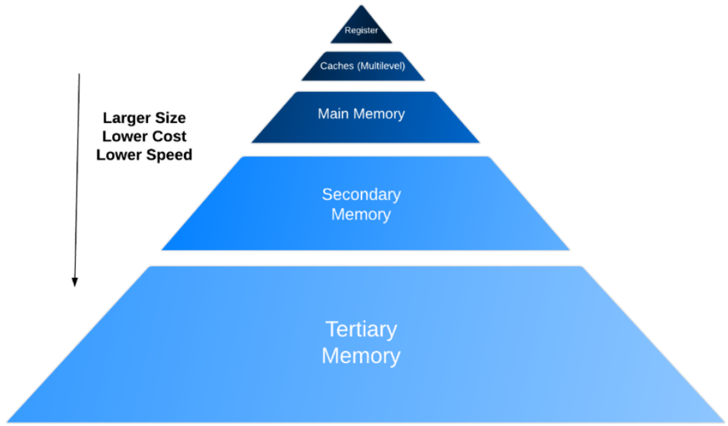
- Program scope: statically allocated
- Function/method scope: automatically allocated on the stack
- From explicit allocation to explicit deallocation: on the heap
- Always define the most restrictive scope
- Use dynamic allocation on the heap with care

Class qualifiers and data type:

Data type	Size	Signed Range	Unsigned Range
char, int8_t, uint8_t	Byte	-128 to 127	0 to 255
short, int16_t, uint16_t	Half word	-32768 to 32767	0 to 65535
int, int32_t, uint32_t, long	Word	-2147483648 to 2147483647	0 to 4294967295
long long, int64_t, uint64_t	Double word	-2^{63} to 2^{63}-1	0 to 2^{64}-1
float	Word	-3.4028234 × 10^{38} to 3.4028234 × 10^{38}	
double, long double	Double word	-1.7976931348623157 ×10^{308} to 1.7976931348623157 ×10^{308}	
pointers	Word	0x00 to 0xFFFFFFFF	
enum	Byte/ half word/ word	Smallest possible data type	
bool (C++), _bool(C)	Byte	True or false	
wchar_t	Half word	0 to 65535	

- `const`: the value of the variable cannot be changed after initialization. Never written by program, can be put in ROM to save RAM.
- `volatile`: the value of the variable can be changed by external events. Can be changed outside of normal program flow: ISR, hardware register Compiler must be careful with optimizations.
- `static`: the variable is shared between all instances of the class. Declared within function or method, retains value between function/method invocations Declared within classes: the field is instantiated once for all class instances and the value is retained for the program lifetime.

Memory hierarchy:



- Register: usually one CPU cycle to access
- Cache: Static RAM
- Main Memory: Dynamic RAM, Volatile data
- Secondary Memory: Flash/Hard disk
- Tertiary Memory: Tape libraries

Bootloader

A Bootloader is a program that runs when a device is powered on or reset. It is responsible for loading the operating system kernel to memory and starting its execution. It is usually stored in the non-volatile memory of the device. Primary purpose: allow a software/firmware to be updated without the use of specialized hardware (No JTAG programmer).