C++ Software Engineering

for engineers of other disciplines

Module 4

"C++ Embedded"

1st Lecture: **`while(true)`**

ALTEN

Summer 2020

Gothenburg, Sweden

rashid.zamani@alten.se

Embedded Systems

ALTEN

- Computer system with a specific function *embedded* as a part of a complete device with a more complex function.
- Given the nature of their functioning environment, they are mostly restricted to *real-time computing* limitations.
- Embedded systems are usually optimized to perform their very specific task to reduce costs and increase reliability and performance.

Developing for Embedded Systems

In general, little is "special" about developing for embedded systems:

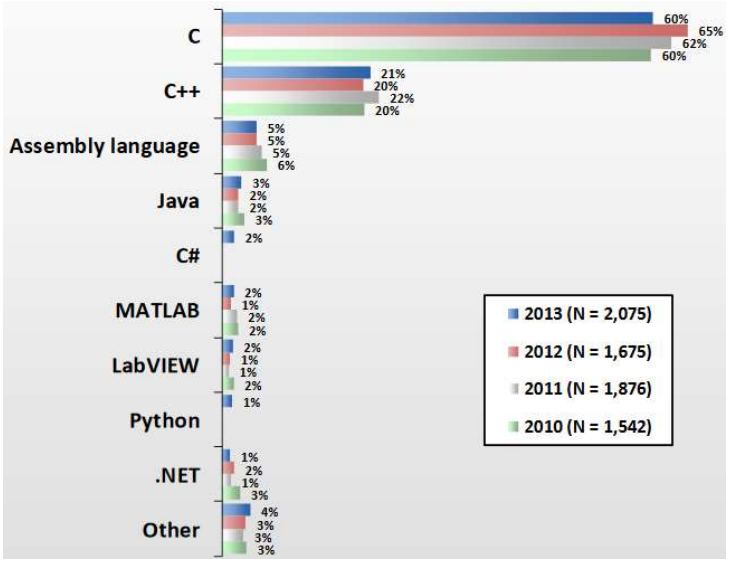
- Software must respect the constraints of the problem and platform.
- C++ language features must be applied judiciously.

These are true for non-embedded applications, too.

- **Good embedded software development is just good software development.**

https://www.artima.com/shop/effective_cpp_in_an_embedded_environment
Presentation Materials: Effective C++ in an Embedded Environment, by Scott Meyers – free sample, slide 9

- C programming language provides possibility of optimization on machine instruction level and given its history and penetration it has had in the industry; it is the most used language in embedded systems and current industry's performance benchmark.



http://images.content.ubmtechelectronics.com/Web/UBMTechElectronics/%7Ba7a91f0e-87c0-4a6d-b861-d4147707f831%7D_2013EmbeddedMarketStudyb.pdf

Embedded Software -- Architecture

ALTEN

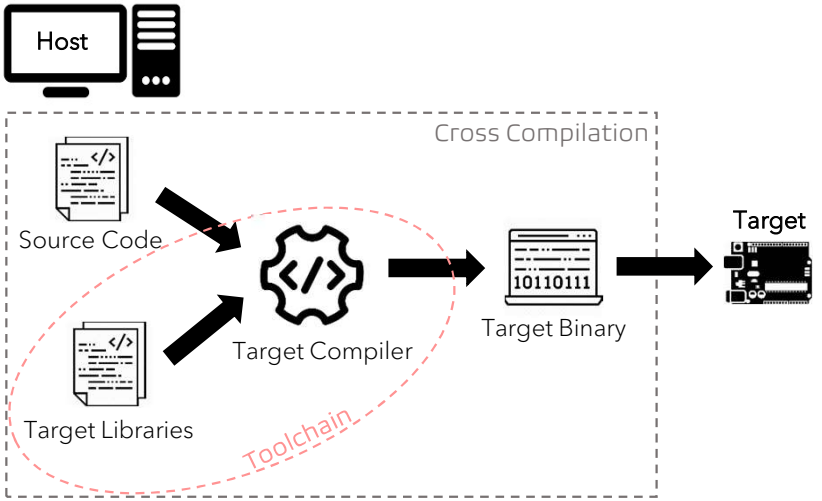
- Embedded systems are often running in a loop:
  - Closed-Loop (automated): This is a loop which happens for *eternity!*
  - Open-Loop (non-automated): This is a loop that halts and awaits confirmation or approval.
- Within architecture of an embedded system, there might be many different loops. The *eternal* loop located in `main` function is called the *"super loop".*
- *Super Loop* is common amongst embedded systems running on *"bare metal"* without an OS.

```
int main() {
  initialize();
  while (system_is_running) {
    check_status();
    perform_operations();
    delay();
  }
}
```

- Initialization step usually includes detecting hardware and loading drivers, a step analogous to OS booting.
- Within each loop, from a very abstract perspective, the status of the system is checked which could reading sensor data or user input. Depending on the status some operations might be performed. And finally, a delay could be calculated prior to the next iteration. This could allow for better resource management by avoiding unnecessary checking.

Embedded Software -- Development

- Embedded systems architectures vary, and they run different operating systems (or even none!).
- Embedded software are expected to be reliable, robust and deterministic.
- Real-time constraints applies to embedded programs and that is to guarantee responses within specified time limits a.k.a. '*deadlines*'.
  - Hard/Immediate real-time systems must meet all the deadlines.
  - Soft/Firm real-time system could miss some deadlines with the cost of degraded performance.
- Only features from C++ should be used which comply with embedded systems' requirements.

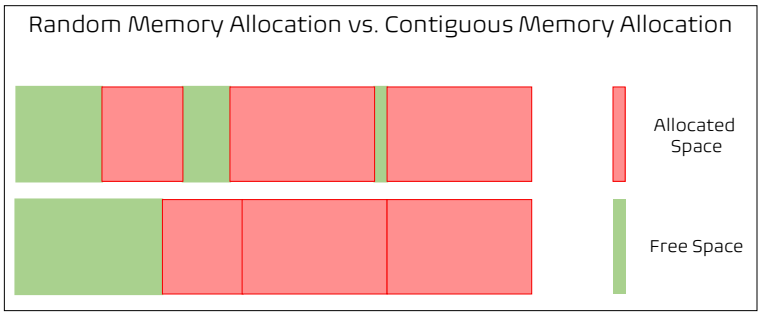


- Due to computational resource restriction and other limitations, in many cases, it is not possible to implement software on the embedded system itself. The software is implemented on different machine a.k.a. the *host*, and then it is compiled on the *host* using the embedded *target* toolchain (compiler and necessary libraries). The result is a binary which could be executed only on the *target*.

Embedded Software -- Challanges

ALTEN

- General Embedded Software Development Concerns:
  - Apart from memory leakage, *dynamic memory* allocation could result in memory fragmentation. As well as allocation delays which are not predictable, also acquiring a space is not guaranteed -- certain data structures are used to guarantee deterministic allocation such as ***pools*** or ***stacks***.
  - *Run-time* operations such as vtable look ups for pure virtual functions, do not provide the necessary reliability, while impose execution delays – stuff which are "*clear*" at compile time could, in theory, assure determinism, correctness, and predectibility.
  - Certain features of the programming languages, if not employed correctly, could result in *code bloat* at machine code level – something hidden from novice programmers looking at source code.



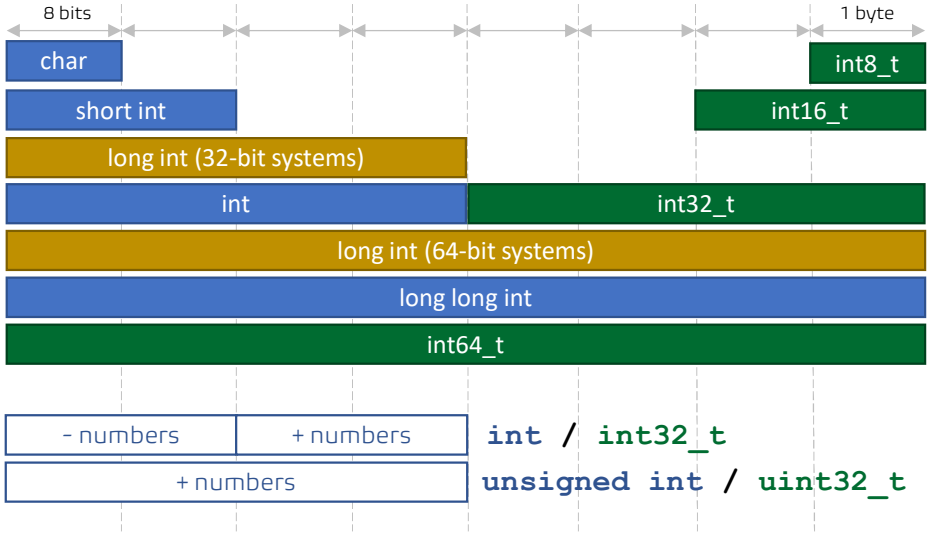
- Memory fragmentation occurs when allocation happens in non-contiguous blocks.
- STL containers by default use new and free either directly or indirectly.

"In computer programming, code bloat is the production of program code (source code or machine code) that is perceived as unnecessarily long, slow, or otherwise wasteful of resources." https://en.wikipedia.org/wiki/Code_bloat

Bit

- Resources are limited in embedded systems, every *bit* counts!
- Binary representation of the data depicts its formation in bits.
- ***Bitwise operations*** are very common in Embedded Systems.
- Some bitwise operation on signed values are compiler dependent or undefined, therefore, unsigned values are use.
- Bytes are also called an *octet* since each byte holds 8 bits. Half of a byte (4 bits) is called a *nibble*.

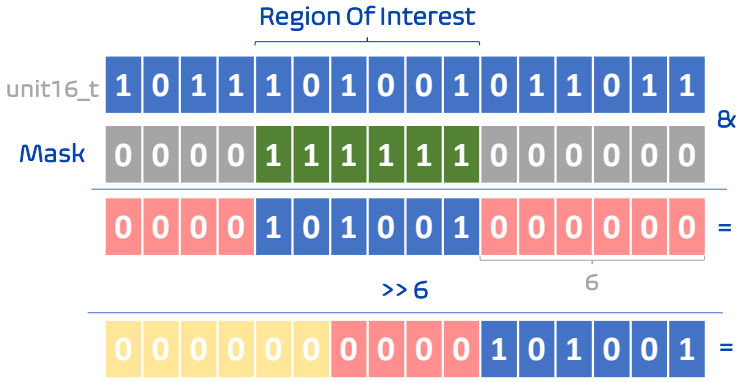
HEX	5	HEX	8	HEX	F	HEX	FF
DEC	5	DEC	8	DEC	15	DEC	255
OCT	5	OCT	10	OCT	17	OCT	377
BIN	0101	BIN	1000	BIN	1111	BIN	1111 1111



- Types could vary in size depending on the architecture and the compiler.
- Range of the numbers between `signed` and `unsigned` numbers of same datatype size varies.

Bitwise Operations

- Operations performed at bit level -- inherited from C.
- Bitwise operations are directly supported by the processors. They are fast and simple – usually used to manipulate values for:
  - Bitwise arithmetic calculations, and
  - Boolean/Logical comparisons.



- A *bitmask* or *mask* is data that is used to manipulated certain bits in a bit sequence or a bit field.
- Boolean size is 1 byte, while each byte could hold 8 Boolean values within each bit.

```
Size of bool is: '1' Byte!
```

```
std::cout << "Size of bool is: \'"
          << sizeof(bool) << "\' Byte!"
          << std::endl;
```

	Bitwise operator	Description
Boolean Operators	&	AND
	|	OR
	^	XOR
	~	NOT
Logical Shift Operators	>>	Shift Right
	<<	Shift Left

a	b	a | b	a & b	a ^ b
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

a	a >> 2	a << 1	a	~ a
0100	0001	1000	0	1
1010	0010	0100	1	0

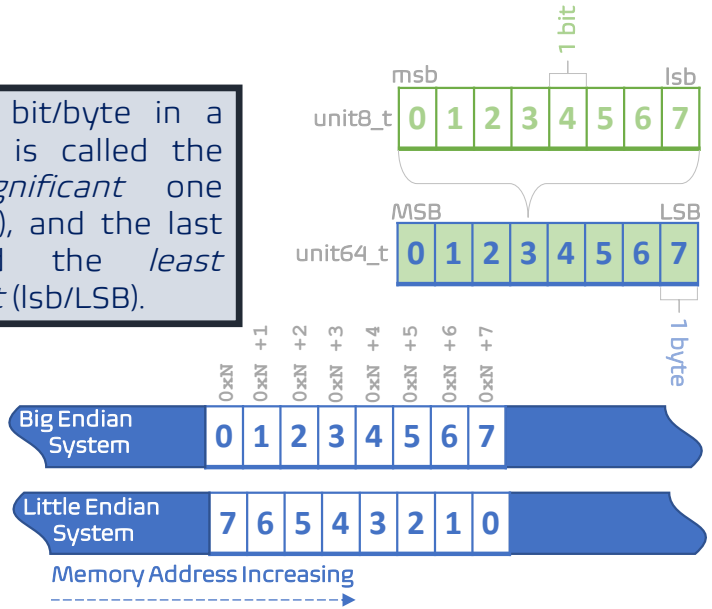
Endianess

- Different systems architecture store ***word***s in different sequences:
  - Big-Endian (BE): MSB is stored in the least memory address.
  - Little-Endian (LE): MSB is stored in the highest memory address.
- Endianness could occur at bit level as well, specially *transmission order* of bits over a serial medium.

> *"A word is a fixed-sized piece of data handled as a unit by the instruction set or the hardware of the processor […] The size of a word is reflected in many aspects of a computer's structure and operation."* https://en.wikipedia.org/wiki/Word_(computer_architecture)

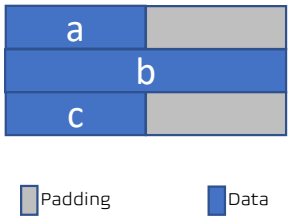
```cpp
int main() {
  union {
    uint64_t i;
    uint8_t c[8];
  } data;
  data.i = 0x0102030405060708;
  if (data.c[0] == 0x01) {
    std::cout << "Big-endian architecture." << std::endl;
  } else if (data.c[7] == 0x01){
    std::cout << "Little-endian architecture." << std::endl;
  } else {
    std::cout << "Does not make sense." << std::endl;
  }
}
```

- The first bit/byte in a sequence is called the *most significant* one (msb/MSB), and the last is called the *least significant* (lsb/LSB).



Padding

- For better performance, data store in memory is ***aligned*** with the size of *word.*
- Compiler inserts *padding* when necessary to *align data.*
- It is important to make sure a good alignment is used for more efficient usage of memory.
- Some programming languages, including C++, provide means for modifying or specifying *data alignment*, to ensure efficient resource usage.




```
struct Bad_Aligend
{
    uint8_t a;
    uint16_t b;
    uint8_t c;
};
struct Good_Aligend
{
    uint8_t a;
    uint8_t c;
    uint16_t b;
};
```

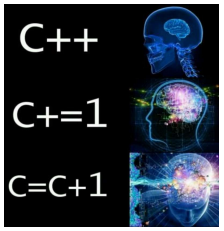
```
sizeof (struct Bad_Aligend) : 6
sizeof (struct Good_Aligend): 4
```

```
int main()
{
    std::cout << "sizeof (struct Bad_Aligend) : " << sizeof (struct Bad_Aligend)  << std::endl;
    std::cout << "sizeof (struct Good_Aligend): " << sizeof (struct Good_Aligend) << std::endl;
    return 0;
}
```

C++ As An Embedded Language

ALTEN

- C++ intends to be a "*better*" C!
- Rich features in C++ are indeed resource-intensive, yet it is guaranteed that:
  - There is zero-overhead rule i.e. you do not pay for what you are not using, if you very well know what you are using!
  - There is no lower-level language below C++ except the assembler, thus your program translates into machine code.
- Extra care should be put into resource usage, determinability, resilience and robustness of the code.



- Costs here are compared to C, and basically are:
  - Size of the code
  - Execution time
  - Memory usage

C++ "concerning" Features	C++ "zero-cost" Features
new/delete	Anything inherited from C
Templates	Namespaces
Exceptions	Static functions & data
Run-time type information	Classes, *Constructors* & *Destructors*
Pure Virtual Functions	Non-virtual & *virtual* member functions
Multiple Inheritances	Function overloading
	Single inheritance
	Virtual inheritance

EC++

ALTEN

- In order to ensure *concerning* features are not employed and the programming language is used *properly*; different *industrial consortia* provide sets of rules, guidelines, tools, and standard on how to use C++.

- These "*guidelines*" may share the same views over many concepts, yet they have different regulations to meet specific requirements of the industry the function in; some examples are:

  - Embedded C++ (EC++)
  - Motor Industry Software Reliability Association (MISRA) C++
  - AUTomotive Open System ARchitecture (AUTOSAR) C++

"Embedded C++ (EC++) is a dialect of the C++ programming language for embedded systems. It was defined by an industry group led by major Japanese central processing unit (CPU) manufacturers [...] to address the shortcomings of C++ for embedded applications [...] a restricted subset of C++ (based on Embedded C++) has been adopted by Apple Inc. as the exclusive programming language to create all I/O Kit device drivers for Apple's macOS, iPadOS and iOS operating systems of the popular Macintosh, iPhone, and iPad products." https://en.wikipedia.org/wiki/Embedded_C%2B%2B

The Embedded C++

FUJITSU JNOVEL Japan Novel Corporation Red Hat NEC HITACHI TOSHIBA

C++ features banned in EC++
Run-time type information
Templates
Exceptions
Style-Casts (C++ way of casting)
Namespaces
Multiple Inheritances
Virtual Inheritances

MISRA

- Motor Industry Software Reliability Association (MISRA), provides a set of software development guidelines to facilitate *code safety*, *security*, *portability*, and *reliability* in the context of embedded system.
- Initiated for C programming language, and later issued guidelines for C++; the guidelines, since introduction, were heavily adopted by industries dealing with critical systems.
- There are two standard released at 2008 & 2012, which are referenced to by other guidelines, like AUTOSAR.
- MISRA guidelines also focuses on software engineering context, as well as programming language and coding context. It also defines steps necessary to make sure an implementation adheres to the subset.
- Guidelines are categorized into three classes: *required rules*, *advisory rules*, and *document* rules.

- Certain tools are available for both static and dynamic analysis to check whether a given source code complies with the MISRA regulation. In some industries MISRA compliance is a requirement.

- MISRA also provide rules, independent of the language. These rules are listed in the below table.

	Rule	Description
Unnecessary Constructs	0-1-1	A project shall not contain unreachable code.
	0-1-2	A project shall not contain infeasible paths.
	0-1-3	A project shall not contain unused variables.
	0-1-4	A project shall not contain non-volatile POD variables having only one use.
	0-1-5	A project shall not contain unused type declarations.
	0-1-6	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.
	0-1-7	The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.
	0-1-8	All functions with void return type shall have external side effect(s).
	0-1-9	There shall be no dead code.
	0-1-10	Every defined function shall be called at least once.
	0-1-11	There shall be no unused parameters (named or unnamed) in non-virtual functions.
	0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.
Storage	0-2-1	An object shall not be assigned to an overlapping object.
Run-time failures	0-3-1	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.
	0-3-2	If a function generates error information, then that error information shall be tested.
Arithmetic	0-4-1 & 2	Use of scaled-integer or fixed-point or floating-point arithmetic shall be documented.
	0-4-3	Floating-point implementations shall comply with a defined floating-point standard.

AUTOSAR

- AUTomotive Open System ARchitecture (AUTOSAR) aims to create and establish an open and standardized software architecture for automotive ECUs to ensure *scalability* and *transferability of software* as well as focusing on *availability* and *safety* requirements.

- AUTOSAR specifies coding guidelines for C++14, in *safety-related* and *critical systems.* MISRA C++ 2008 is a prerequisite for AUTOSAR C++14 – most guidelines have been adopted from MISRA *"as-is"*, while there are some which have been modified or dismissed.

- AUTOSAR C++14 rules are mostly enforceable by static analysis. However, for some of the rules there need to be a manual review.

- AUTOSAR C++14 could be viewed as a more relaxed standard compared to the ones MISRA provides. For instance, dynamic memory usage is allowed in AUTOSAR while banned in MISRA C++.

- It has been announced that MISRA and AUTOSAR will publish one standard for C++17 in a joint effort.
- AUTOSAR C++14 is targeting *high-end* embedded system that provide efficient and full C++14 support on 32- and 64-bit systems.

AUTOSAR

ALTEN

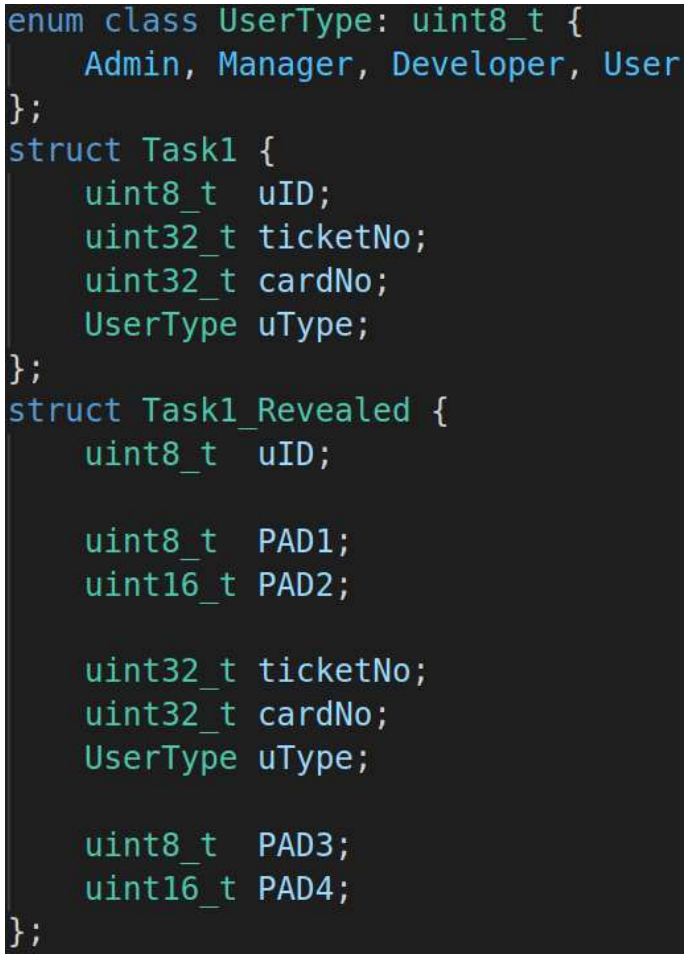
- AUTOSAR provides guidelines for some of the concerning features of C++ to ensure they are used in the correct (less risky) manners. However, there are some features which their usage is banned.
- AUTOSAR provides tools and instructions for both static and dynamic analysis of the software to verify its level of compliance with the guidelines. Simplicity of the code is something which should be measured in AUTOSAR as well.

C++ features which shall not be used
malloc and free
C-style casts
const_cast
dynamic_cast
reinterpret_cast
goto statement
Return type deduction
typedef specifier
asm declaration
Variadic arguements
Unions
Multiple inheritance
friend declaration
User-defined literals
Function-try-blocks
Dynamic exception specification
#pragma directives

C++ features which could be used under AUTOSAR C++ 14 guidelines	
Dynamic memory management	List initialization
Floating-point arithmetic	Bit-fields
Operators new and delete	Inheritance
Sized deallocation	Virtual Functions
Namespaces	override specifier
Fixed width integer types	final specifier
nullptr pointer literal	Defaulted and deleted functions
static_cast	Delegation constructors
Lambda expressions	Member initializer list
Generic lambda expression	Non-static member initializer
Binary literals	explicit specifier
Range-based for loops	Move semantics
constexpr specifier	Digit sequence separators
auto specifier	Variadic templates
decltype specifier	Variable templates
Trailing return type syntax	Exceptions
using specifier	noexcept specifier
Scoped enumerations	Static assertion
std::initializer_list	
Default arguments	

DEMO!

Alignment

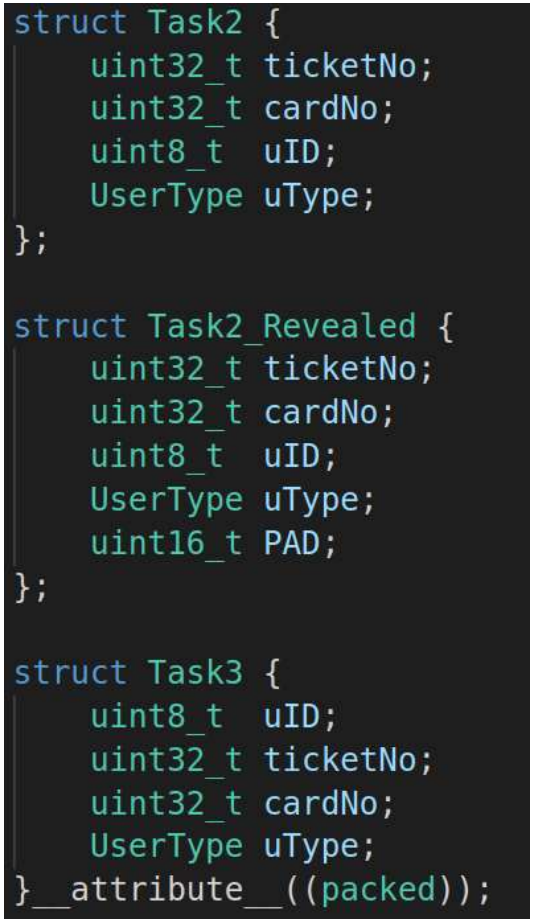


```
enum class UserType: uint8_t {
    Admin, Manager, Developer, User
};
struct Task1 {
    uint8_t  uID;
    uint32_t ticketNo;
    uint32_t cardNo;
    UserType uType;
};
struct Task1_Revealed {
    uint8_t  uID;

    uint8_t  PAD1;
    uint16_t PAD2;

    uint32_t ticketNo;
    uint32_t cardNo;
    UserType uType;

    uint8_t  PAD3;
    uint16_t PAD4;
};
```



```
struct Task2 {
    uint32_t ticketNo;
    uint32_t cardNo;
    uint8_t  uID;
    UserType uType;
};

struct Task2_Revealed {
    uint32_t ticketNo;
    uint32_t cardNo;
    uint8_t  uID;
    UserType uType;
    uint16_t PAD;
};

struct Task3 {
    uint8_t  uID;
    uint32_t ticketNo;
    uint32_t cardNo;
    UserType uType;
} __attribute__((packed));
```

Bitfield

ALTEN

```
struct field {
    unsigned  b1:1;
    unsigned  b2:2;
    unsigned  b3:3;
    unsigned  b4:4;
}__attribute__((packed));
```

```
void printField(const field &_f) {
    std::cout << "----------------" << std::endl;

    std::cout << "(int)_f.b1 " << (int)_f.b1 << std::endl; //BAD LAZY CAST
    std::cout << "(int)_f.b2 " << (int)_f.b2 << std::endl;
    std::cout << "(int)_f.b3 " << (int)_f.b3 << std::endl;
    std::cout << "(int)_f.b4 " << (int)_f.b4 << std::endl;
    std::cout << "----------------" << std::endl;

}
```

```
field a = {0xFF,0xFF,0xFF,0xFF},b,c[8],*d;

b.b1 = 2;
b.b2 = 2;
b.b3 = 2;
b.b4 = 2;

std::cout << sizeof(field) << std::endl;
std::cout << sizeof(a) << std::endl;
std::cout << sizeof(c) << std::endl;
```

```
uint16_t Data[5] = { 0xffff, 0xffff,0xffff,0xffff,0xffff};
std::memcpy(c, Data, 5*sizeof(uint16_t));

for (size_t i = 0; i < 8; i++) {
    printField(c[i]);
}
std::cout << "----------------" << std::endl;
d = reinterpret_cast<struct field*>(Data);
for (size_t i = 0; i < 8; i++) {
    printField(*(d+i));
}
```

Bitset

```cpp
std::string bits = "10101100";
std::bitset<8> set1(bits);
std::bitset<8> set2(255);

std::cout << "set1\t" << set1 << std::endl;
std::cout << "set2\t" << set2 << std::endl;
std::cout << "(~set2)\t" << (~set2) << std::endl;

std::cout << "(set1 << 2)\t" << (set1 << 2)  << std::endl;
std::cout << "(set2 >> 1)\t" << (set2 >> 1)  << std::endl;
std::cout << "(set1 & set2)\t" << (set1 & set2) << std::endl;
std::cout << "(set1 | set2)\t" << (set1 | set2) << std::endl;
std::cout << "(set1 ^ set2)\t" << (set1 ^ set2) << std::endl;
```

Bitmask


```
unsigned char MASK = 0b00111100, Data[2] = {0b10100100,0b10110101};
std::cout << std::bitset<8>(MASK) << " ^\t" <<
             std::bitset<8>(Data[0]) << " = " <<
             std::bitset<8>(Data[0] ^ MASK) << std::endl;

std::cout << std::bitset<8>(MASK) << " |\t" <<
             std::bitset<8>(Data[1]) << " = " <<
             std::bitset<8>(Data[1] | MASK) << std::endl;

std::cout << std::bitset<8>(MASK) << " &\t" <<
             std::bitset<8>(Data[0]) << " = " <<
             std::bitset<8>(Data[0] & MASK) << std::endl;

std::cout << std::bitset<8>(MASK) << " &\t" <<
             std::bitset<8>(Data[1]) << " = " <<
             std::bitset<8>(Data[1] & (~MASK)) << std::endl;

std::cout << std::bitset<8>(MASK) << " &>>2\t" <<
             std::bitset<8>(Data[0]) << " = " <<
             std::bitset<8>( (Data[0] & MASK) >> 2) << std::endl;
```