CS301
Embedded System and Microcomputer Principle

Lecture 3: C for Embedded System

2024 Fall

Recap

- GPIO
  - CPU
    - ALU
    - Registers
      - R1~R12
      - SP, LR, PC
      - PSR (flags)
    - Control unit
  - GPIO
    - Memory mapped IO
      - CPU treats peripherals registers the same as data memory
    - GPIO Mode
      - default: floating input
      - input: pull up/pull down
      - output: push-pull
    - Registers
      - 1 port -> 16 pins -> 7 registers
  - Programming GPIO
    - 1.Enable GPIO Clock
      - RCC->APB2ENR
    - 2. Configure GPIO mode
      - CRL (pin0~pin7)
      - CRH (pin8~pin15)
      - ODR (if input, 1: pull up, 0: pull down
    - 3a. Output value setting
      - ODR
    - 3b. Input value reading
      - IDR

Embedded vs Desktop Programming

- Main characteristics of embedded programming environments:
  - Cost sensitive
  - Limited ROM, RAM, stack space
  - Limited power
  - Limited computing capability
  - Event-driven by multiple events
  - Real-time responses and controls
  - Reliability
  - Hardware-oriented programming

Embedded Programming

- Basically, optimize the use of resources:
  - Execution time
  - Memory
  - Energy/power
  - Development/maintenance time
- Time-critical sections of program should run fast
  - Processor and memory-sensitive instructions may be written in assembly
  - Most of the codes are written in a high level language (HLL): e.g. C, C++, or Java

Outline

- **Operations**
- Data Types
- Storage Classes

Arithmetic

- Integer arithmetic → Fastest
- Floating-point arithmetic in hardware → Slower
- Floating-point arithmetic in software → Very slow

+, −

×

÷

sqrt, sin, log, etc.

slower

- Try to use integer addition/subtraction
- Avoid multiplication unless you have hardware
- Avoid division
- Avoid floating-point, unless you have hardware
- Really avoid math library functions

Bit Manipulation

- C has many bit-manipulation operators:

  & Bit-wise AND

  | Bit-wise OR

  ^ Bit-wise XOR

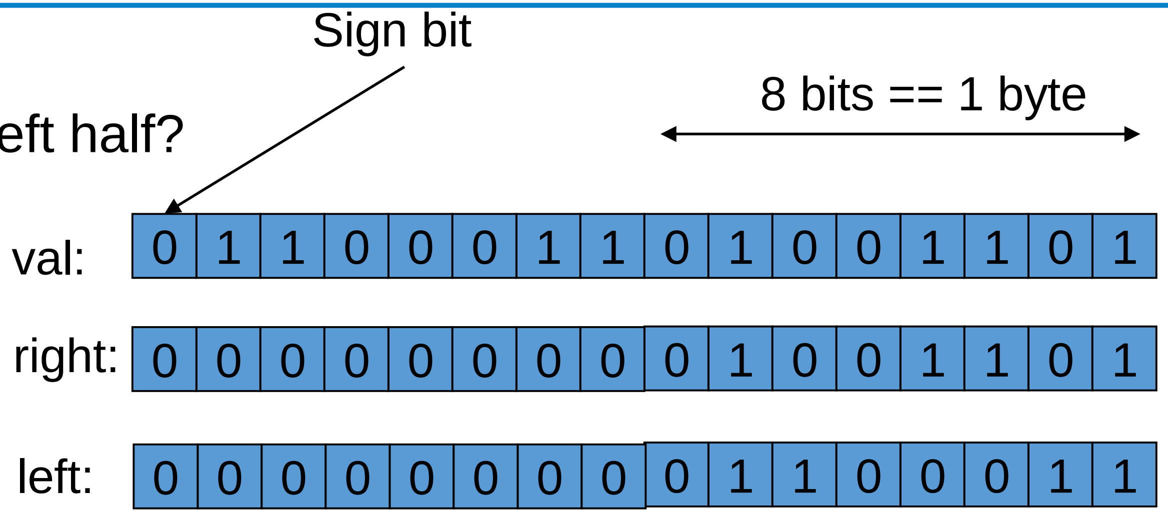
  ~ Negate (one’s complement)

  >> Right-shift

  << Left-shift
- Plus assignment versions of each: &=, |=, etc
- Often used in embedded systems

Bit Manipulation

- Example
  - val = 0x634B, how to get right half and left half?




```
void f(unsigned short val)      //assume 16-bit, 2-byte unsigned short integer
{
    unsigned short right = val & 0x00ff;
    // rightmost (least significant) byte
    unsigned short left = (val>>8) & 0x00ff;
    // leftmost (most significant) byte
    bool negative = val & 0x8000;
    // sign bit (if 2's complement)
    // …
}
```

C doesn’t have booleans until C99 standard. Bool emulate as int or char, with values 0 (false) and 1 or non-zero (true)

Bit Manipulation in STM32

- use | operator to set a bit of a byte to 1

```
GPIOA->ODR |= (1<<8); //set bit 8 (9th bit) of GPIOA->ODR
```

- use & operator to clear a bit in a byte to 0

```
GPIOB->ODR &= ~(1<<8); //clear bit 8 (9th bit) of GPIOB->ODR
```

- use & operator to see if a bit in a byte is 1 or 0

```
if( ((GPIOC->IDR & (1<<5)) != 0)  //check bit 5 (6th bit)
```

A mask indicates which bit positions we are interested in

Faking Multiplication/Division

- Addition, subtraction, and shifting are fast
  - Can sometimes supplant multiplication
- Like floating-point, not all processors have a dedicated hardware multiplier
  - Multiplication is realized by addition and subtraction
  - Multiplication to a power of two is just a shift
- Division is a much more complicated algorithm that generally involves decisions
  - Division by a power of two is just a shift:

    a / 2 = a >> 1

    a / 4 = a >> 2

Lazy Logical Operators

- “Short circuit” tests save time

  `if (a == 3 && b == 4 && c == 5) { ... }`
- equivalent to

  `if (a == 3) { if (b ==4) { if (c == 5) { ... }`
- Strict left-to-right evaluation order provides safety

  `if ( i < SIZE && a[i] == 0 ) { ... }`

Multi-way branches

- Which one is faster? Shorter?

```
if (a == 1)
    foo();
else if (a == 2)
    bar();
else if (a == 3)
    baz();
else if (a == 4)
    qux();
else if (a == 5)
    quux();
else if (a == 6)
    corge();
…
```

```
switch (a) {
case 1:
    foo(); break;
case 2:
    bar(); break;
case 3:
    baz(); break;
case 4:
    qux(); break;
case 5:
    quux(); break;
case 6:
    corge(); break;
}
…
```

Code for if-then-else

```c
if (a == 1)
    foo();
else if (a == 2)
    bar();
else if (a == 3)
    baz();
else if (a == 4)
    qux();
else if (a == 5)
    quux();
else if (a == 6)
    corge();
…
```

```
    20: void test(int a) {
0x080004DC B500      PUSH     {lr}
    21:         if (a == 1)
0x080004DE 2801      CMP      r0,#0x01
0x080004E0 D102      BNE      0x080004E8
    22:         foo();
0x080004E2 F7FFFFCF  BL.W     foo (0x08000484)
0x080004E6 E017      B        0x08000518
    23: else if (a == 2)
0x080004E8 2802      CMP      r0,#0x02
0x080004EA D102      BNE      0x080004F2
    24:         bar();
0x080004EC F7FFFF89  BL.W     bar (0x08000402)
0x080004F0 E012      B        0x08000518
    25: else if (a == 3)
0x080004F2 2803      CMP      r0,#0x03
0x080004F4 D102      BNE      0x080004FC
    26:         baz();
0x080004F6 F7FFFF85  BL.W     baz (0x08000404)
0x080004FA E00D      B        0x08000518
    27: else if (a == 4)
0x080004FC 2804      CMP      r0,#0x04
0x080004FE D102      BNE      0x08000506
    28:         qux();
0x08000500 F7FFFFEB  BL.W     qux (0x080004DA)
0x08000504 E008      B        0x08000518
    29: else if (a == 5)
0x08000506 2805      CMP      r0,#0x05
0x08000508 D102      BNE      0x08000510
    30:         quux();
0x0800050A F7FFFFE5  BL.W     quux (0x080004D8)
0x0800050E E003      B        0x08000518
    31: else if (a == 6)
0x08000510 2806      CMP      r0,#0x06
0x08000512 D101      BNE      0x08000518
    32:         corge();
0x08000514 F7FFFF77  BL.W     corge (0x08000406)
    33: }
```

Code for Switch

```
switch (a) {
case 1:
    foo(); break;
case 2:
    bar(); break;
case 3:
    baz(); break;
case 4:
    qux(); break;
case 5:
    quux(); break;
case 6:
    corge(); break;
…
}
```

BCS: Branch if Carry Set
TBB: Table Branch Byte
DCW: allocates a half-word

```
    35: void testswitch(int a) {
0x080004DC B500      PUSH     {lr}
    36:         switch(a){
    37:         case 1:
0x080004DE 2807      CMP      r0,#0x07
0x080004E0 D217      BCS      0x08000512
0x080004E2 E8DFF000  TBB      [pc,r0]
0x080004E6 0416      DCW      0x0416
0x080004E8 0A07      DCW      0x0A07
0x080004EA 100D      DCW      0x100D
0x080004EC 0013      DCW      0x0013
    38:                 foo(); break;
    39:         case 2:
0x080004EE F7FFFFC9  BL.W     foo (0x08000484)
0x080004F2 E00E      B        0x08000512
    40:                 bar(); break;
    41:         case 3:
0x080004F4 F7FFFF85  BL.W     bar (0x08000402)
0x080004F8 E00B      B        0x08000512
    42:                 baz(); break;
    43:         case 4:
0x080004FA F7FFFF83  BL.W     baz (0x08000404)
0x080004FE E008      B        0x08000512
    44:                 qux(); break;
    45:         case 5:
0x08000500 F7FFFFEB  BL.W     qux (0x080004DA)
0x08000504 E005      B        0x08000512
    46:                 quux(); break;
    47:         case 6:
0x08000506 F7FFFFE7  BL.W     quux (0x080004D8)
0x0800050A E002      B        0x08000512
    48:                 corge(); break;
0x0800050C F7FFFF7B  BL.W     corge (0x08000406)
0x08000510 BF00      NOP
```

Computing Function x=f(a)

- There are many ways to compute a "random" function of one variable, especially for sparse domain:

```
if (a == 0) x = 0;
else if (a == 1) x = 4;
else if (a == 2) x = 7;
else if (a == 3) x = 2;
else if (a == 4) x = 8;
else if (a == 5) x = 9;
…
```

- Better for large, dense domains, switch cases use a jump table

```
switch (a) {
    case 0: x = 0; break;
    case 1: x = 4; break;
    case 2: x = 7; break;
    case 3: x = 2; break;
    case 4: x = 8; break;
    case 5: x = 9; break;
…
}
```

- Best: constant time lookup table

```
int f[] = {0, 4, 7, 2, 8, 9};
x = f[a]; /* assumes 0 <= a <= 5 */
```

Preprocessor and Macro

- The preprocessor is executed before the compilation. Main usages:
  - File inclusion
  - Macro substitution
  - Conditional compilation
  - preprocessing instruction: #include, #ifdef, #ifndef, #if, #else, #define, etc
- Macro is a fragment of code that is given a name and can be used as a replacement for that code in the source code

```c
#define SQUARE(x) x * x
int area(int r)
{
    return SQUARE(r);
}
```

Preprocess

```c
int area(int r)
{
    return r * r;
}
```

Function vs Macro

- A named collection of codes
  - A function is compiled only once. On calling that function, the processor has to save the context, and on return restore the context
  - Preprocessor puts macro code at every place where the macro-name appears. The compiler compiles the codes at every place where they appear.
- Function versus macro:
  - Time: use function when $T_{overheads} << T_{exec}$, and macro when $T_{overheads}$ ~= or > T_{exec}, where $T_{overheads}$ is function overheads (context saving and return) and T_{exec} is execution time of codes within a function
  - Space: similar argument

Outline

- Operations
- **Data Types**
- Storage Classes

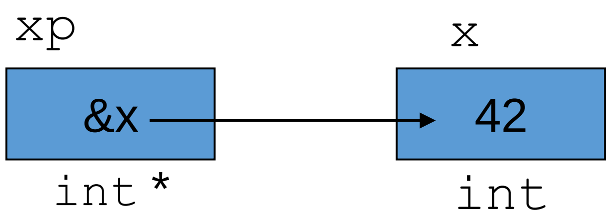
Data Type

Type	# bit	Range	Stdint type (stdint.h)	ST type (stm32f10x.h)
char	8	-128 ~ 127	int8_t	s8
unsigned char	8	0 ~ 255	uint8_t	u8
short	16	-32768 ~ 32767	int16_t	s16
unsigned short	16	0 ~ 65535	uint16_t	u16
int	32	-2147483648 ~ 2147483647	int32_t	s32
unsigned int	32	0 ~ 4294967295	uint32_t	u32
long	32	-2147483648 ~ 2147483647		
unsigned long	32	0 ~ 4294967295		
long long	64	-(2^64)/2 ~ (2^64)/2-1	int64_t	
unsigned long long	64	0 ~ (2^64)-1	uint64_t	
float	32	-3.4e38 ~ 3.4e38		
double	64	-1.7e308 ~ 1.7e308		

Pointers

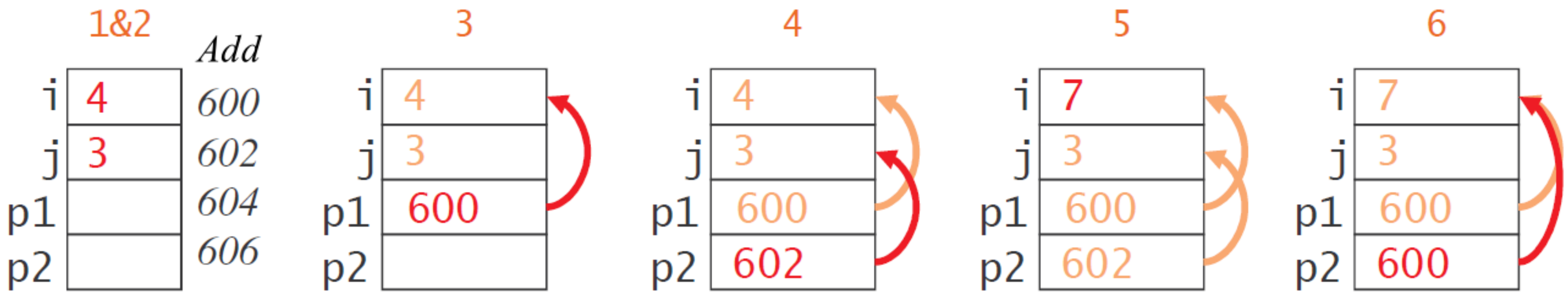
- A **pointer** is the memory **address** of a data object
- e.g.

  int x = 42;
  int *xp = &x;




```
void main ( ) {
  int i, j;
  int *p1, *p2;
1 i = 4;
2 j = 3;
3 p1 = &i;
4 p2 = &j;
5 *p1 = *p1+*p2;
6 p2 = p1;
}
```

  - The pointer xp holds the address of x
  - The data x is accessed by dereferencing the pointer xp using *xp.
  - Pointer's data size is usually 4 or 8 byte
  - Dangling pointer = danger !



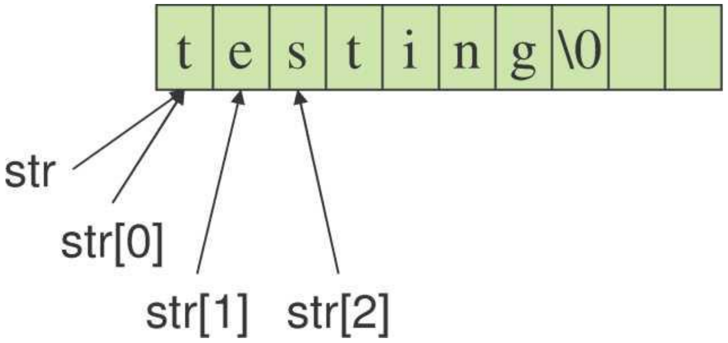
Pointer vs Array

- Incrementing and decrementing pointers to array elements
  - Increment operator ++ makes pointer advance to next element
  - Decrement operator -- makes pointer move to previous element
  - These use the size of the variable’s base type (e.g. int, char, float) to determine what to add
    - p++ (Equivalent to p=p+1) increments the pointer p by the size of the data type it points to, for example sizeof(int)
    - sizeof is C operator which returns size of type in bytes

```
int a[18];
int * p;
p = &a[5];
*p = 5; /* a[5]=5 */
p++;
*p = 7; /* a[6]=7 */
p--;
*p = 3; /* a[5]=3 */
```

Pointer vs String

- There is no “string” type in C.
- Instead, an array of characters is used: char a[44];
- The string is terminated by a NULL character (value of 0, represented in C by \0).
  - Need an extra array element to store this null
- Example
  - char str[10] = “testing”;




```c
char str[30];
int a = 30;
float b = 3.14;

sprintf(str, "a=%d, b=%f\n", a, b);

sprintf(str, "a=0x%X, b=%.1f\n", a, b);
```

str

a=30, b=3.140000

str

a=0x1E, b=3.1

Structure

- a structure is an aggregate data type composed of several distinct members.

```c
struct PERSON {
    char gender;
    int age;
};

struct PERSON sue;
```

or

```c
typedef struct {
    char gender;
    int age;
} PERSON;

PERSON sue;
```

- GPIOC->CRL
  - GPIOC_BASE: base address of GPIOC registers
  - GPIOC: pointer to a GPIO_TypeDef structure
  - GPIOC->CRL: access the CRL register of GPIOC

```c
#define GPIOA_BASE            (APB2PERIPH_BASE + 0x00000800UL)

/** @addtogroup Peripheral_declaration
  */
#define GPIOB                 ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC                 ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD                 ((GPIO_TypeDef *) GPIOD_BASE)
```

Cast to pointer type

```c
/** @brief General
  *Purpose I/O
  */
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

Memory Alignment

- Modern processors have byte-addressable memory
  - But, many data types (integers, addresses, floating-point) are wider than a byte
  - In 32-bit system, data are transferred in 32-bit chunks
- Structures member are stored in order, but memory aligned

```
Struct {
     char gender;
     int age;
} sue;
```

Extra bytes added so that member ‘age’ begins on a mod 4 (word) address

31			0
padding	*padding*	*padding*	*sue.gender*
sue.age			

= Added padding

3	2	1	0
7	6	5	4
11	10	9	8

```
struct padded {
     int x;    /* 4 bytes */
     char z;   /* 1 byte */
     short y;  /* 2 bytes */
     char w;   /* 1 byte */
};
```

x	x	x	x
y	y		z
			w

Structure Bit Fields

- Aggressively packs data to save memory
  - Compiler will pack these fields into words
  - Implementation-dependent packing, ordering, ...
  - Usually not very efficient in terms of execution time: requires masking, shifting, read-modify-write
    → a tradeoff between space and time!

```
struct PSR // Assumes bitfields are allocated from LSB to MSB
{
    uint32_t exception : 9, // Current exception number
        : 1,                // (reserved)
        ici_it1 : 6,        // (see technical reference manual)
        : 8,                // (reserved)
        T : 1,              // Always 1
        ici_it2 : 2,        // (see technical reference manual)
        Q : 1,              // Saturation flag
        V : 1,              // Signed overflow flag
        Z : 1,              // Zero or equal flag
        C : 1,              // Unsigned carry flag
        N : 1;              // Negative flag
};
```

C Unions

- Like structs, but shares the same storage space and only stores the most-recently-written field

```
union {
    int ival;
    float fval;
    char *sval;
} u;
```

  - but occupy same memory space
  - can hold different types at different times
  - overall size is largest of elements
  - Potentially very dangerous: not type-safe

Data Type Selection

- Mind the architecture
  - Same C source code could be efficient or inefficient
  - Should keep in mind the architecture's typical instruction size and choose the appropriate data type accordingly
- 3 rules for data type selection:
  - Use the smallest possible type to get the job done
  - Use unsigned type if possible
  - Use casts within expressions to reduce data types to the minimum required
- Use typedefs to get fixed size
  - Change according to compiler and system
  - Code is invariant across machines

```c
/* Fixed-size types */
typedef unsigned char uint8_t;
typedef short int16_t;
typedef unsigned int uint32_t;
```

Strength Reduction

- Why multiply when you can add?

```c
struct {
    int a;
    char b;
    int c;
} foo[10];
int i;
for (i = 0; i < 10; ++i) {
    foo[i].a = 77;
    foo[i].b = 88;
    foo[i].c = 99;
}
```

```c
struct {
    int a;
    char b;
    int c;
} *fp, *fe, foo[10];
fe = foo + 10;
for (fp = foo; fp != fe; ++fp) {
    fp->a = 77;
    fp->b = 88;
    fp->c = 99;
}
```

- Good optimizing compilers automatically do this

Outline

- Operations
- Data Types
- **Storage class**

Storage class in C

- keywords related to storage class provide information on
  - **scope**: visibility or accessibility
  - **lifetime**: existence of a variable

```c
/* fixed address: visible to other files */
extern int global_static;
/* fixed address: visible within file */
static int file_static;
/* parameters always stacked */
int foo(int auto_param)
{
    /* fixed address: only visible to func */
    static int func_static;
    /* stacked: only visible to function */
    int auto_i, auto_a[10];
    /* array explicitly allocated on heap */
    double *auto_d = malloc(sizeof(double) * 5);
    /* return value in register or stacked */
    return auto_i;
}
```

Storage class in C

- auto:
  - Default storage class for variables declared inside a function.
  - Accessible only within the function's scope.
- register:
  - Stores variables in CPU registers to speed up access.
  - Same scope and accessibility as auto; allocation depends on the compiler.
- static:
  - Variable lifetime matches the program’s duration.
  - Retains its value between function calls; initialized to zero by default.
  - Accessibility depends on declaration context.
- extern:
  - Makes variables accessible across multiple functions and files.
  - Can be modified by any function within its scope.

Static

- Static variable inside function is initialized only once.
- What's the value of y?

```c
int foo();
int main(void) {
   int y;
   y = foo();  // y = ?
   y = foo();  // y = ?
   y = foo();  // y = ?
   while(1);
}
int foo() {
   int x = 5;
   x = x + 1;
   return(x)
}
```

```c
int foo();
int main(void) {
   int y;
   y = foo();  // y = ?
   y = foo();  // y = ?
   y = foo();  // y = ?
   while(1);
}
int foo() {
   static int x = 5;
   x = x + 1;
   return(x)
}
```

Extern

- See how extern allows access to the global variable count across different files

```c
#include <stdio.h>

int count ;
extern void write_extern();

int main()
{
   count = 5;
   write_extern();
}
```

```c
#include <stdio.h>

extern int count;

void write_extern(void)
{
   printf("count is %d\n", 
count);
}
```

Volatile

- A volatile variable is one whose value may be change outside the normal program flow. In embedded systems, there are two ways this can happen:
  - Via an interrupt service routine
  - As a consequence of hardware action
- It is considered to be very good practice to declare all peripheral registers in embedded devices as volatile
  - Modify the variable with the volatile keyword so that the value of the variable is re-read every time instead of using the backup value stored in the register

```c
volatile int flag = 0;
void f() {
    while (1) {
        if (flag) {
            do_action();
        }
    }
}
```

```c
void isr_f0() {
    flag = 1;
}
```

malloc() and free()

- Flexible than (stacked) automatic variables
- More costly in time and space
- Use non-constant-time algorithms
- Two-word overhead for each allocated block:
  - Pointer to next empty block
  - Size of this block
- Common source of errors:

  Using uninitialized memory　　Using freed memory
  Not allocating enough　　Indexing past block
  Neglecting to free disused blocks (memory leaks)

Good or bad for embedded applications?

Storage Compared

- On most processors, access to automatic (stacked) data and globals is equally fast
  - Automatic usually preferable since the memory is reused when function terminates
  - Danger of exhausting stack space with recursive algorithms. Not used in most embedded systems.
- The heap (malloc) should be avoided if possible:
  - Allocation/deallocation is unpredictably slow
  - Danger of exhausting memory
  - Danger of fragmentation
  - Best used sparingly in embedded systems

Summary

1. Integer arithmetic
2. Pointer access
3. Simple conditionals and loops
4. Static and automatic variable access
5. Array access
6. Floating-point with hardware support
7. Switch statements
8. Function calls
9. Floating-point emulation in software
10. Malloc() and free()
11. Library functions (e.g. sin, log, etc)
12. Operating system calls (e.g. open)