CPSC 2310 Exam 2 Study Guide

Good luck today! You're gonna do great!

Representing Integers (Signed vs. Unsigned)

For signed bits, use the first bit to represent the sign of the number. This means that signed representations have half the range of their unsigned counterparts.

The range for unsigned data is 0 to $(2^w - 1)$

The range for signed data is (-2^{w-1}) to $(2^{w-1} - 1)$

Ways to Represent Signed Integers

Sign-Magnitude. The value of the integer bits added together, except for the most significant bit, the most significant bit tells you the sign of the number (1 = negative, 0 = positive)

1's Complement. If num is positive (MSB is 0), add up the values. If negative (MSB is 1), complement the bits and then add them.

2's Complement. MSB acts as a negative value. Complement all the bits and add 1 to negate.

Minimum and Maximum

w = word size	**Minimum Value**	**Maximum Value**
Unsigned	0	2^{w-1}
Signed	-2^{w-1}	$2^{w-1} - 1$

Expanding Bit Representation

When casting from a lower information type to a higher information type, extension can be either zero or signed. Higher representation bits are added (the MSB changes) to fill in the empty space.

Zero Extension (unsigned numbers). In unsigned types, higher representation bits are filled with zeros.

```
(uint_8t) 0b1000 0000 => (uint_16t) 0b0000 0000 1000 0000
```

Signed Extension (signed numbers). For signed numbers, we check the MSB to preserve the sign. If the MSB is 1 fill with ones, else fill with zeros

```
(int_8t) 0b1000 0000 => (int_16t) 0b1111 1111 1000 0000
(int_8t) 0b0000 1000 => (int_16t) 0b0000 0000 1000 0000
```

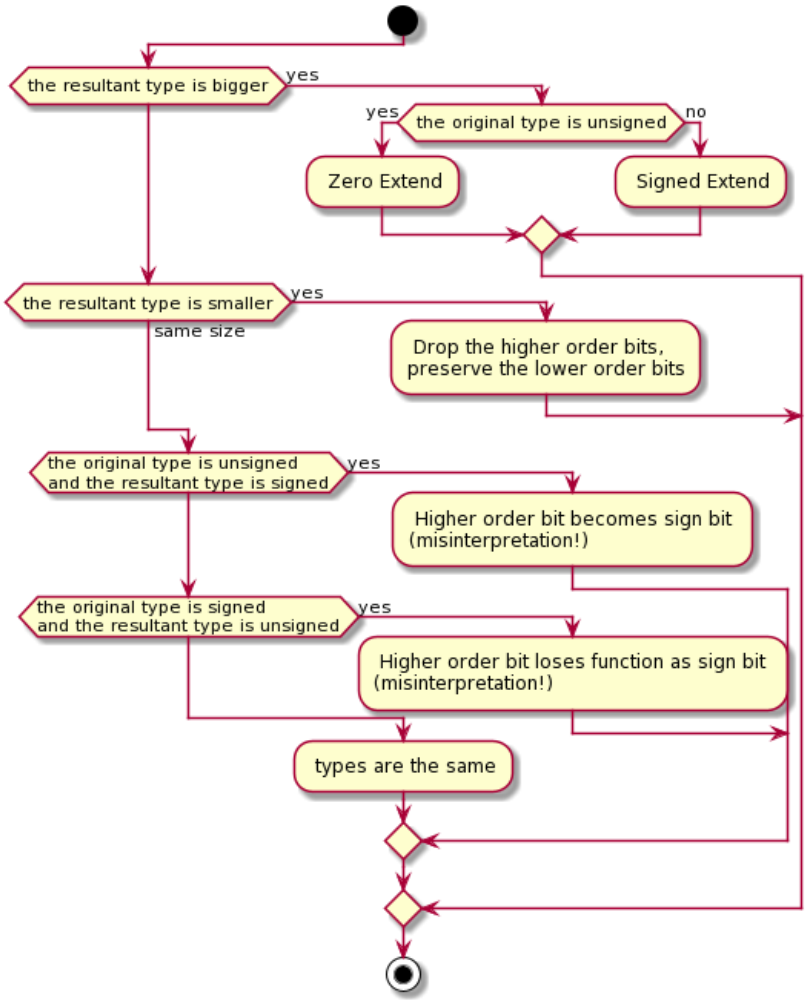
Reducing Number of Bits (Truncation)

When dropping bits to fit the number into a smaller representation, drop the higher bits and keep the lower order bits. *This can lead to information being misinterpreted!* See below for the conversion table.

Conversion Table

char = unsigned char Preserve bit pattern; high-order bit becomes sign bit
short = unsigned char Zero-extend
long = unsigned char Zero-extend
unsigned short = unsigned char Zero-extend
unsigned long = unsigned char Zero-extend
char = unsigned short Preserve the low-order byte
short = unsigned short Preserve bit pattern; high-order bit becomes sign bit
long = unsigned short Zero-extend
unsigned char = unsigned short Preserve low-order byte
char = unsigned long Preserve low-order byte
short = unsigned long Preserve low-order byte
long = unsigned long Preserve bit pattern; high-order bit becomes sign bit
unsigned char = unsigned long Preserve low-order byte
unsigned short = unsigned long Preserve low-order byte

Type Conversion Rules of Thumb



Signed & Unsigned Comparison

By default, C handles comparison expressions (`==`, `<`, `>`) as signed, unless you explicitly cast them to unsigned in the comparison (or are comparing an unsigned signed variable). In that case, *both expressions will get interpreted as unsigned numbers! This can lead to some strange edge cases!*

Left Side	Comparison	Right Side	Interpreted As	Result
0 0000 0000	==	0U 0000 0000	Unsigned	1
-1 1111 1111	<	0 0000 0000	Signed	1
-1 1111 1111	<	0U 0000 0000	Unsigned 1111 1111 = 255	0 INCORRECT!
127 0111 1111	>	-127 - 1 1000 0000	Signed	1
127U 0111 1111	>	-127 - 1 1000 0000	Unsigned 1000 0000 = 128	0 INCORRECT!

Overflow

Integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside the range of that can be represented, either too high (overflow) or too low (underflow).

Truncation Error - When casting from a larger integer type to a smaller integer type, because the higher-order bits are truncated, this can lead to issues.

Unsigned Addition

When $x + y > 2^w - 1$, the sum overflows. Perform the addition as standard, and then truncate the top bits to make the result fit inside. If the sum of the two numbers is less than either of the original numbers, then an overflow has occurred.

```
uint_8t a = 0b1111 0000
uint_8t b = 0b0001 1111
uint_8t c = a + b // 0b0001 0000 1111 => 0b0000 1111
if (b > c || a > c) {
  printf("Overflow!")
}
```

Unsigned Negation

For a number, the unsigned negation is the number you can such that it will overflow to exactly zero. This is the complement of the bits of the number.

```
uint_8t a = 0b1111 0101
uint_8t b = 0b0000 1011
uint_8t c = a + b // 0b0001 0000 0000 = 0b0000 0000
```

Two's Complement Addition

Can have negative overflow (sum is less than the tmin) or positive overflow (sum is greater than tmax). Negative overflow occurs when sum >= 0 and x < 0 and y < 0. Positive overflow occurs when x > 0, y > 0, s <= 0

```
uint_8t a = 0b1111 0101 // -128 + 64 + 32 + 16 + 4 + 1 = -11
uint_8t b = 0b0011 1010 // 32 + 16 + 8 + 2 = 58
uint_8t c = a + b // 0010 1111 = 47
```

Two's Complement Negation

Every number in the range has an additive inverse. This is just the two's complement (flip the bits and add one).

```
uint_8t a = 0b0010 1001 // 32 + 8 + 1 = 41
uint_8t b = 0b1101 0111 // -128 + 64 + 16 + 4 + 2 + 1 = -41
uint_8t c = 0b0000 0000 // The 9th bit is 1, but gets truncated, so 0
```

Unsigned Multiplication

Requires 2x as many bits to store the product of two numbers. *For example, a uint_8t * uint_8t requires uint16_t.* For x and y such that 0<= x,y <= $Umax_w$ = (x*y) mod 2^w

Traditionally, multiplication was computationally expensive (10ish clock cycles), so compilers will optimize multiplication by replacing it with constant shifts and addition.

Multiplication Optimizations

To multiply a number by shifts, determine the powers of two that sum to the number, and left shift the number by each, and then add (or subtract) them together.

```
let x = 3 * 55
```

$55 = 2^6 - 2^3 - 2^1$

```
3 = 0000 0011
0000 0011 << 6 =  1100 0000
0000 0011 << 3 =  0001 1000
0000 0011 << 1 =  0000 0110

1100 0000 + 0000 0110 - 0001 1000
= 1100 1110 - 0001 1000
= 1011 0110
```

Overflow in Initialization

Remember that numeric literals in C are `int`s, so assignments can *cause a cast operation!*

```
// (int) 235 = 0000 0000 0000 0000 0000 0000 1110 1011 → 1110 1011 = (int8_t) -21
int8_t variable = 235;
variable == -21;
```

Overflow in Bitshifts

Bit shifts cannot be more than the word size of the type you are shifting by. *For example, you cannot shift ints by 32 or greater. If you do, the compiler will take the mod 32 of the amount to shift.*

```
int8_t variable 0b1000 1000;
variable >> 9; // actually 9 mod 8 = 1; // 0b1100 0100;
```

C Topics

Function Pointers

Function pointers allow you to store the memory address of a function in a variable, allowing you to swap the specific function being called at runtime. This can be used for many applications but is commonly used in implementing callbacks. Type information about the function is stored in the function pointer type.

```c
// Creates a type definition for the function pointer (makes working with it
// easier)
typedef int (*operation)(int a, int b);

int add(int a, int b) {
  return a + b;
}

int multiply(int a, int b) {
  return a * b;
}

// Use the typedef above to make the function prototype a little more readable
// Could also be written without the typedef as
// int operate(int a, int b, int (*operation)(int, int))
int operate(int a, int b, operation op) {
  return op(a, b);
}

int main() {

  int a = 4;
  int b = 5;

  int sum = operate(a, b, &add); // 4 + 5 = 9
  int product = operate(a, b, &multiply); // 4 * 5 = 20

  // Can be called without &
  int sum = operate(a, b, add); // 4 + 5 = 9
  int product = operate(a, b, multiply); // 4 * 5 = 20

}
```

Structures

Structures let you group multiple data types together into a single unit. In the below example, we are optimizing the size of the struct with bit fields, restricting int to be smaller than typically expected.

```
typedef struct Date {
  unsigned int date : 5 // 5 bits is enough to represent 1-31
  unsigned int month : 4 // 4 bits is enough to represent 1-12
  unsigned int year
} date_t;

int main() {

  struct Date p;
  struct Date *pointer = &p;
  date_t p2;

  // Initialize p
  p.date = 1;
  p.month = 4;
  p.year = 2021;

  // Initialize p2 using designated initializer, a C-only feature (not in C++).
  // Notice how field names can be out of order
  p = {
    .year = 2021,
    .month = 5,
    .date = 21
  }

  // Initialize exam date with just an initializer list (no named fields)
  date_t exam = { 8, 3, 2021 };

}
```

Unions

Unions are very similar to structures, however, instead of storing multiple distinct data types, all values of a union *share the same memory location*. This can be useful for more direct type conversion/reinterpretation.

```c
#include <stdio.h>

/**
 * This union allows for easy serialization and deserialization of an integer to
 * an array of bytes, allowing it to be transmitted more easily
 */
typedef union {
    int number;
    unsigned char bytes[4];
} serializable_int;

int main() {

    serializable_int a;
    a.number = 63533;

    printf("%d => ", a.number);
    for (int i = 0; i < 4; i++) {
        printf("%d ", a.bytes[i]);
    };

    printf("\n");
};
```

Macros

Macros are instructions to the preprocessor to modify the *source code* of your file before processing further. This can be incredibly powerful in allowing you to shorten repetitive parts of your program, but can also be quite dangerous if not handled with care!

```
#define MAX_SIZE 10
// From now on, the preprocessor will replace MAX_SIZE with 10

int array[MAX_SIZE];

// Undefine MAX_SIZE, we can no longer use it
#undef MAX_SIZE

// Predefined Macros
__DATE__ // The date you invoked the compiler Mmm dd YYYY
__FILE__ // The current source file
__LINE__ // The line number of the current source file
__TIME__ // The time you invoked the compiler (hh:mm:ss)
__func__ // The name of the function (predefined identifier)

// Function Macros
#define ADD(a, b) a + b
```

Linked Lists

Linked Lists are a simple dynamically-expanding data collection of items. Each item in the list points to the next item in the list, with the last element having a null pointer. In C, linked lists are usually implemented as structs with a pointer field.

Advantages

- ➔ Simple
- ➔ Can easily grow and shrink

Disadvantages

- ➔ Memory is not contiguous; not cache friendly
- ➔ Element access is O(n), computationally expensive
- ➔ Pointer overhead

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

typedef struct Node {
    char ch;
    struct Node *next;
} node_t;

node_t *node_create(char ch){

    node_t *node = (node_t *)malloc(sizeof(node_t));
    node->ch = ch;
    node->next = NULL;

    return node;
};

void node_append(node_t *head, node_t *node){

    node_t *current = head;

    // Iterate through the linked list till you get to the end
    while(current->next != NULL) {
        current = current->next;
    };

    // Set the last node's next to the given node
    current->next = node;
};
```

```c
void node_print(node_t *node){
    printf("0x%.2X (%p)", node->ch, node->next);
    if (node->next != NULL) {
        printf(" -> ");
    };
};

typedef void (*node_callback)(node_t *node);
void node_iterate(node_t *head, node_callback callback) {

    node_t *current = head;

    callback(head);
    while (current->next != NULL) {
        current = current->next;
        callback(current);
    };
}

int main() {

    char string[100];
    printf("Enter a string: ");
    scanf("%99[^\n]", string);

    // Create the head node from the first character in the string
    node_t *head = node_create(string[0]);

    int length = strnlen(string, 100);
    for (int i = 1; i <= length; i++) {
        node_t *node = node_create(string[i]);
        node_append(head, node);
    };

    // Iterate through and print each node
    node_iterate(head, node_print);
    printf("\n");
}
```

Scanset

Scansets allow you to restrict the types of characters that *scanf* will process when reading from an input.

The general format for scansets is as follows, in this order!

Symbol(s)	Required?	Meaning
%	Yes	Start of the scanset
*	No	Do not capture the result of the scanset to a variable
<conversion specifier>	Yes	Instruction on how to interpret the input, including characters to read. Specifies the required type of the pointer passed to the variadic arguments.

Common Conversion Specifiers See `man scanf` for more information		
d	Signed Integer	`int*`
u	Unsigned Integer	`unsigned int*`
c	Unsigned char	`char*`
s	String	`char*`
x	Unsigned Hexadecimal Integer	`unsigned int*`
`[<chars>]`	Matches a nonempty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to char, and there must be enough room for all the characters in the string, plus a terminating null byte. If you have a circumflex (^), the set will exclude all included characters.	`char*`

```c
int main() {
    char str[100];
    printf("Enter a string in lowercase: \n");
    scanf("%[a-z]", str);
    printf("Enter a string in uppercase: \n");
    scanf("%[A-Z]", str );
    printf("Enter a string with a combination of upper, lower, and numbers: \n");
    scanf("%[a-z,A-Z,0-9]", str );
    printf("Enter a string with a upper, lower, numbers and spaces: \n");
```

```
    scanf("%[a-z,A-Z,0-9,' ']",str);
    printf("Enter a string with a combination numbers: \n");
    scanf("%[0-9]",str);
    printf("Enter a string terminated with a return: \n");
    scanf("%[^\n]", str);
    printf("String: %s\n", str);

    char a[15];
    char b[15];
    char c[15];
    char d[15];

    printf("Enter 4 sets of chars,nums, separated with - : \n");
    scanf("%[^-]%*c%[^-]%*c%[^-]%*c%[^\n]%*c", a, b, c, d);
    printf("Strings: %s, %s, %s, %s\n", a, b, c, d);

    return 0;
}
```