Some Details of C and C Environment

In this last lecture, we look at

- Bit orders and storage alignment,
- Advanced aspects of C operators, and
- Some aspects of numerical calculations.

Little and big endian systems I

Suppose we have the definition

```
unsigned int n = 257;
```

Normally n takes 4 bytes. In this case, two bytes contain the number 1 and the other two only zero bits. Is the situation in these Linux machines

100	101	102	103
1	1	0	0

(little endian)

or

100	101	102	103
0	0	1	1

(big endian)?

We can test this by defining

Little and big endian systems II

```
unsigned int n = 257;
uint8_t u_ptr;

u_ptr = &n;

printf("Mem addr of n: %lu\n", u_ptr);
printf("First byte: %d  ", *u_ptr);
printf("Second byte: %d \n ", *(u_ptr+1));
```

If 1 and 1 are printed, the system is little endian. If the system is big endian, maybe zeros are printed or the system crashes (segmentation fault).

Bit order in a byte I

Assume the definitions

```
uint8_t byte = 1;
```

Is the byte now

```
1000 0000
```

or

```
0000 0001 ?
```

We can examine this with shift operations. If the first were the case, then

```
byte >> 1;
```

would produce the value 2. If the latter, then

```
byte << 1;
```

would produce 2. It turns out that the second alternative happens in Linux machines.

Constructing indices I

- The bit order in a byte is important for example, when calculating indices from a byte for AES SubBytes module. The encryption and decryption usually take place in different machines which must treat the bit patterns in the same way in order to decrypt succesfully the enciphered message.
- In the same way the little endian and big endian question is important in network protocols with which different machines communicate.
- Assume that a byte contains 84. So the bit pattern is 01010100. The left part represent the index i, and the right side the index j. Thus $i = 5$, $j = 4$. The values are constructed with the following commands:

Constructing indices II

```
uint8_t byte;
uint8_t mask = 0XF;

j = mask & byte;
i = byte >> 4;
```

Storage alignment I

Consider the following structure definition:

```
typedef struct {
    short Data1;
    short Data2;
    short Data3;
    short Data4;
} Data;
```

Then `sizeof(Data) = 8`. let's consider another structure

```
typedef struct {
    char Data1;
    short Data2;
    int Data3;
    char Data4;
} MixedData;
```

Storage alignment II

The size of MixedData is 8, but the actual size is 10. So padding bytes are added by the compiler.

In Linux systems, malloc reserves memory blocks so that the number of the first byte is divisible by four. However, it seems possible to store an integer so that it start from a byte not divisible by four. This kind of integer position may, however, slow down an execution.

Operators I

We deal with the following:

- Assignment combinations.
- Lazy evaluation.
- Evaluation order and side-effect operators.
- Conditional operator.

Operators: Assignment I

All the assignment-combination operators have the same very low precedence and associate right to left. consider for example the expression

```
t /= n -= m *= k += 7;
```

The evaluation starts with $k = k + 7$ and continues from right to left. So $+$ is made before $*$ even if in normal expressions the precedence of multiplication is higher than the precedence of addition.

Operators: Lazy evaluation I

- With lazy evaluation, when we skip, we skip the right operand.
- This is not confusing, when the right operand is only a simple variable. However, sometimes it is an expression with several operators.
- Consider an example:

```
y = a < 10 || a >= 2 * b && b != 1;
```

  The left operand is `a<10` and the right operand is `a>=2*b && b!=1`.

- If $a = 7$, we skip all after $||$ and y get the value 1. If $a = 17$ and $b = 20$ the expression after && is not evaluated and y becomes 0.
- A usual assumption is that && is executed first, because it has higher precedence. Although precedence controls the construction of the parse tree, precedence simply is not considered when the tree is evaluated.

Operators: Lazy evaluation II

- Sometimes lazy evaluation can substantially improve the efficiency of a program. A much more important use of for skipping is to avoid evaluating parts of an expression that would cause machine crashes or other kinds of troubles.
- Assume that we want to divide a number x, compare the answer to a minimum value, and do an error procedure if the answer is less than the minimum. But it is possible for x to be 0 and that must be checked. We can avoid a division-by-0 error and do the computation and comparison in one expression by using a *guard* before the division:

  ```
  if (x != 0) && total / x < minimum) do_error();
  ```

- The skipping may happen in the middle of an expression. Consider

  ```
  y = a < 0 || a > b && b > c || b > 10;
  ```

  for $a = 3$ and $b = 17$. The subexpression $b > c$ is not evaluated, all the other are and y becomes 1.

Operators: Evaluation Order and Side Effects I

- When used in isolation, the increment and decrement operators are convenient and relatively free of complication. When side-effect operators are used in long, complex expressions, they create the kind of complexity that fosters errors. If such an operator is used in the middle of a logical expression, it may be executed sometimes but skipped at other times.
- A second problem with side-effect operators relates to the order in which the parts of an expression are evaluated. Recall that the evaluation order has nothing to do with precedence order. We have stated that logical operators are executed left to right. This is also true of two other kinds of sequencing operators: the conditional operator ?...: and the comma (for example, for $(i = 1,\ \ j = 1; ...)$). Most other operators can be evaluated right-side first or left-side first.

Operators: Evaluation Order and Side Effects II

- This leads to one important warning: If an expression contains a side-effect operator that changes the value of a variable V, *do not use* V anywhere else in the expression. The side-effect could happen either before or after the value of V is used elsewhere in the expression and the outcome is unpredictable.

Operators: Conditional Operator I

- Even if the evaluation in an expression with the conditional operator starts by calculating true or false value, the value of the entire conditional operator, in general, will not be true or false.
- If the condition contains any postincrement operators, the increments must be done before evaluating the true clause or the false clause. Therefore, it is "safe"to use postincrement in the condition.

I Numerical Calculations I

- Choosing the proper data type.
- Representational properties.
- Computational issues.
- Casts and mixed-type operations.
- Sticky points and common errors.
- Representation error.

Numeric: Choosing the proper data type I

- *Integer or real?* For most problems obvious.
- *Float vs. double*. Because a programmer can combine types float and double freely in expressions, most of the time, it does not matter which real type is used. Sometimes the degree of precision required for the data dictates the use of double. Since all the functions in the math library expect double arguments and return double results, most programmers just find it easier to declare all real varaibles as double.

  However, if you are processing large amounts of data and precision is not important, then float variables use only half as much space as doubles and an int even less in some cases. Integer calculations are faster than real calculations and float values are faster than double values.

Numeric: Representational properties I

- For integers: signed, unsigned, short and long.
  - Max short signed: 32767
  - Max long signed: 2 147 483 647
  - Max long unsigned: 4 294 967 295
- For float the minimum value range is

$$1.175\mathrm{E}-38...3.402\mathrm{E}+38.$$

For double:

$$2.225\mathrm{E}-308...1.797\mathrm{E}+308.$$

Numeric: Computational issues I

- Integer division is not the same as division using real numbers; any remainder from an integer division is forgotten. The remainder must be computed by using a modulus operator(%).
- Sometimes the underlying computer hardware does not support floating-point arithmetic, in which case floating point representation and computation must be emulated by software. This is slow.

Numeric: Casts I

- C supports mixed-type arithmetic. When two values of differing types are used with an operator, the value with less precision automatically is coerced to the more precise representation.
- If an integer is combined with float or a double in an expression, the integer operand always is converted to the type of the floating-point operand before the operation is performed. The result is a floating-point value.
- A type conversion may be safe, in that it will cause no floss of information, or it may be unsafe. Knowing when a type conversion can be used safely is important. However, sometimes an unsafe conversion is exactly what the programmer needs.
- An explicit type cast must be used to perform real division with integer operands.

Numeric: Sticky points and common errors I

- Using the wrong conversion specifier in a format can cause input or output to appear as garbage. Default length, short, and long integers have different conversion codes, as do signed and unsigned integers.
- When using reals, there is no way to tell from the printed output whether a value came from a double or a float variable. If you specify a format such as %.10f, you migght see 10 nonzero digits printed, but that does not mean that all are accurate. If the number came from float variable, the eighth through tenth digits usually will be garbage.
- Some systems use 2 bytes to represent int, others use 4 bytes. This makes the portability of code a nightmare. Errors due to integer sizes are among the hardest to find because of the ever-present automatic size conversions all C translators perform.

Numeric: Representation error I

Types float and double are *approximate representations* for real numbers, but with differing precision. Consider the code below:

```
float w = 4.4;    double x = 4.4;

printf(" Is x == (double) w? %i \n", (x == (double)w) );
printf(" Is (float)x ==  w? %i \n", ((float)x == w) );
```

The output may be unexpected, if you forgot that the two numbers are represented with limited, and different, precision and that the == operator tests for exact bit-by-bit equality:

```
Is x == (double)w? 0
Is (float)x == w? 1
```

Numeric: Representation error II

When a more-precise value is cast to the less-precise type, the extra bits are truncated and the numbers are exactly equal. When a shorter value is cast to the longer type, it is lengthened by adding zero bits at the end of the mantissa, not by recomputing the additional bit values. In general, these zeros are not equal to the meaningful bits in the double value.

Computation also can introduce representational error, as shown by the next code fragment:

```
float w;
  double x, y = 11.0, z = 9.0;

  x = z * (y / z);
  w = y-x;
```

Numeric: Representation error III

The result is as expected, $w = 0$, $x = 11.000$. But change the starting values $y = 15.0$ and $z = 11.0$, and the result is

$$w = 1.77635e - 15, \qquad x = 15.000.$$

Why does this happen? The answer to a floating-point division has a fractional part that is represented with as much precision as the hardware will allow. However, the precision nis not infinite and there is a tiny amount of truncation error after most calculations. Therefore, the answer to y/z may have error in it, and that error is increased when we multiply by z.

Numeric: Making meaningful comparisons I

- When are two floating-point numbers equal? The answer is that they should be called equal if both are approximations for the same real number, even if one approximation has more precision than the other.
- Practical problems often require comparing a calculated value to a specific constant or setpoint or comparing two calculated values that should be equal. Such a comparison is not as simple as it seems, because even simple computations with small floating-point values can have results that differ from the mathematically correct versions.
- We can get around this comparison problem by comparing the difference of the two numbers to a preset epsilon value. For any given application, we can choose a value of epsilon that is slightly smaller than the smallest measurable difference in the data.
- This kind of comparison can be made with a single if commmand, if we use *fabs()* function from the math library:

Numeric: Making meaningful comparisons II

```
double epsilon = 1.0e-3;
double number, target;

if (fabs( number -target) < epsilon)
   /* then we consider that number == target */
else
   /* we consider the values different; */
```

Numeric: Overflow I

- **Integer overflow and wrap**. Suppose that the 2-byte integer variable k contains the number 32300 and you enter a loop that adds 100 to k seven times. The value stored in k would be, in turn, 32400, ..., 32700, -32736, ..., -32536. The value has wrapped around and become negative, but that does not stop the computer. This can be a potential security hole allowing buffer overflow attacks.
- **Floating-point overflow and infinity**. The phenomenon of wrap is unique to integers; floating-point overflow is handled differently. The IEEE floating-point standard defines a special bit pattern, called infinity, that will result if overflow occurs during a computation. The constant HUGE_VAL, defined in math.h, is set to be the infinity value on each local system. Therefore, one way an overflow can be detected is by comparing a result to HUGE_VAL or -HUGE_VAL.

Numeric: Overflow II

- **Underflow**. Underflow occurs, when the magnitude of the number falls below the smallest number in the representable range. This can, of course, happen only for real numbers. For real numbers, underflow happens when a value is generated that has a 0 exponent and a nonzero mantissa. Such a number is referred to as *denormalized*, which means that all significant bits have been shifted to the right and the number is less than the lowest number specified by the standard. Some systems will generate the 0 value when the lower bound has been reached. Others still use the denormalized values. Underflow can result from several kinds of computations:
  - Dividing a number by a very large number or repeated division.
  - Multiplying a small number by a near-zero number, which has the same effect as dividing by a very large number.
  - Subtracting two values that are near the smallest representable float and ought to be equal but are not quite equal because of round-off error.

Numeric: Orders of Magnitude and Other Problems I

- If you add a small float number to a large one, and their exponents differ by more than 10^7, the addition likely will have no effect. The answer will be the same large number that you started with.
- A special value called **NAN**, which stands for "not a number", can be generated through operations such as 0/0. This is another special bit pattern that does not correspond to a real value. The IEEE standard defines that any further operation attempted using a NaN or Infinity as an operand will return the same value.

Programming in C, course exam October 20, 2011 I

1. Analyse the following pieces of code where pointers are used. Explain verbally and with the help of diagrams, what happens and if the code is working properly or not. If it is not working, explain the error and correct the code. Diagrams should show the memory locations of the variables, the contents of those locations, and to what locations pointers refer in every phase.

   a)

```
double * p1;
double * p2;

p1 = malloc(sizeof(double));
*p1 = 3.112;
p2 = p1;
printf("%f",*p2);
```

Programming in C, course exam October 20, 2011 II

b)

```
char * s1 = "abcdefghijklm";
char * s2;

s2 = s1;
printf("%s",s2);
```

c)

```
char * s1 = "abcdefghijklmn";
char * s2;

strcpy(s2,s1);
printf("%s",s2);
```

d) Consider the following structure:

Programming in C, course exam October 20, 2011 III

```
struct person{
    char * name;
    int age;
};
```

The following piece of code reads n names and ages from the keyboard, creates a structure for every name and places the names and ages into the corresponding structures.

```
struct person * ptr;
int i;
char  name[50];
int age;

ptr = malloc(n*sizeof(struct person));
/* n defined earlier */
```

```
for (i=0; i < n; i++)
{
    fgets(name,50,stdin);
    scanf("%d", &age);
    (ptr+i)->name = name;
    (ptr+i)->age = age;
}
```

Is the code working properly?

2. Write a function

```
int read_data(char * filename, int * max_line);
```

that opens the file given as a a parameter and calculates the number of lines in the file. Empty lines are counted, too. In addition, the function calculates the length of the longest line and returns it with the help of the other parameter. Lines may be arbitrary long.

Programming in C, course exam October 20, 2011 V

3. Write a function

```
uint16_t make_16bit(uint8_t least_significant,
                    uint8_t most_significant);
```

that combines the given bytes into one 16-bit integer and returns the result.
Example:

```
print_bin(1) => 00000000.00000000.00000000.00000001
print_bin(7) => 00000000.00000000.00000000.00000111
print_bin(make_16bit(7, 1)) =>
                00000000.00000000.00000001.00000111
```

4. a) Let p be a pointer i a variable of integer type. What does $p + i$ mean?
   b) Let p and q be pointers. How do you intepret $p - q$? When is the expression erroneous?

Programming in C, course exam October 20, 2011 VI

c) Let p be a pointer and a an array. Where does p refer to after $p = a$ and after $p = \&a[0]$?

d) Let a be an array. What does $*(a + i)$ mean?

e) Suppose that `sizeof(short) == 2` and `sizeof(int) == 4`. Consider an array `int arr[5] = { 0, 0, 0, 0, 0 };` Explain where the following statement stores values (both the location and the amount of bytes written).

```
((short*)arr)[7] = 128;
```

ATTACHMENT:

The definitions of some C functions:

Programming in C, course exam October 20, 2011 VII

```
int scanf ( const char * format, ... );
int fgetc ( FILE * stream );
int fscanf ( FILE * stream, const char * format, ... );
char * fgets ( char * str, int num, FILE * stream );
FILE * fopen ( const char * filename, const char * mode );
int fclose ( FILE * stream );
char * strcpy ( char * destination, const char * source );
size_t strlen ( const char * str );
void * malloc ( size_t size );
void * calloc ( size_t num, size_t size );
```

OBS: Give some feedback of the course, either anonymously using the www feedback system of the department or manually on a separate paper when your return yourn programming work.

Programming in C, course/separate exam, January 17, 2012 I

1. Consider the following code fragments, where pointers are used. Explain with the help of words and drawings, what happens in various phases. The drawings should show the contents of the memory locations of variables and pointers after commands. Moreover, if there are print commands, what are printed?

   a)

```
int a;
int *aPtr;

a = 7;
aPtr = &a;
printf("%d\n", &a);
printf("%d\n", aPtr);
```

Programming in C, course/separate exam, January 17, 2012 II

```
printf("%d\n", a);
printf("%d\n", *aPtr);
printf("%d\n", &*aPtr);
printf("%d\n", *&aPtr);
```

b)

```
char * s1 = "abcdefghijklm";
char * s2;

s2 = s1;
printf("%s",s2);
```

Programming in C, course/separate exam, January 17, 2012 III

c)
```
void cubeCalculation(int *ptr)
{
      *ptr = *ptr * *ptr * *ptr;
}

In the main program:

int number = 5;
cubeCalculation(&number);
printf("%d", number);
```

Programming in C, course/separate exam, January 17, 2012 IV

2. Consider the data structure in the appendix representing an automaton:
   Write a function `int unreachable(Automaton * A)` which examines, if there are from the initial state unreachable states. The function returns 0, if there are no unreachable states, otherwise 1.
3. Write a function

   ```
   int read_data(char * filename);
   ```

   that open the file given as a a parameter and calculates the number of lines in the file. Empty lines are counted, too. Lines may be arbitrary long.
4. Explain how function parameters are used. Give a simple example.