1 Binary and Hexadecimal Numbers

2 Basic Data Types
   2.1. int Type .................................................. 3
   2.2. char Type ................................................. 7
   2.3. float/double Type ....................................... 8

3 Programmer-Defined Types
   3.1. typedefs .................................................. 12
   3.2. struct Types ............................................. 12
   3.3. enum Types ............................................... 14

4 Working With Types
   4.1. Type Checking ............................................ 15
   4.2. Type Inference ............................................ 16
   4.3. Type Conversion ......................................... 17
   4.4. Type Casting ............................................. 19
1. Binary and Hexadecimal Numbers

- The **type** of a variable specifies
  - the meaning of the variable’s value
  - how the variable’s value should be stored in the computer
  - what operations are allowed on the variable

- Critical to keep concept of **types** separate from concept of **values**

- C is a **statically typed** language, meaning that the type of a variable must be know at compile time

- Keep in mind that no matter how complex the type, everything is ultimately stored as a binary number in the computer

1. Binary and Hexadecimal Numbers

Let’s review decimal, binary, and hexadecimal number representations.
2. Basic Data Types

2.1. int Type

Covert the following decimal number into a binary and hexadecimal number:

300

Covert the following binary number into a hexadecimal and decimal number:

01101110

2. Basic Data Types

We will primarily use the following primitive C types:

• int: For representing signed and unsigned integer numbers
• char: For representing characters
• float/double: For representing real numbers

2.1. int Type

• Meaning? Integer whole number
• Stored? 32-bit two’s complement binary representation
• Operations? Basic integer arithmetic
2. Basic Data Types

2.1. int Type

```c
int avg(int x, int y)
{
    int sum = x + y;
    return sum / 2;
}

int main()
{
    int a = 10;
    int b = 20;
    int c = avg(a, b);
    return 0;
}
```
2. Basic Data Types

2.1. int Type

Signed vs. Unsigned Integers

- By default, an int is short-hand for the type signed int which can represent both positive and negative integers
- unsigned int can only represent positive integers

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Two’s Comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-3</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>

1. signed int a = 4;
2. signed int b = -1;
3. unsigned int a = 4;
4. unsigned int b = 4294967295;

Overflow and Underflow

- Assume we have a signed 4-bit binary number
- Can store values from -8 to 7
- Adding one to 7 in a four-bit binary number = overflow
- Subtracting one from -8 in a four-bit binary number = underflow
• An int is a signed 32-bit binary number
• Can store values between -2,147,483,648 to 2,147,483,647
• What happens if you add one to 2,147,483,647?
• What happens if you subtract one from -2,147,483,648?

• An unsigned int is an unsigned 32-bit binary number
• Can store values from 0 to 4,294,967,295
• What happens if you add one to 4,294,967,295?
• What happens if you subtract one from 0?

```c
#include <stdio.h>

int main()
{
    int a = 2147483647;
    int b = a + 1;
    printf("%d (%x)\n",b,b);

    int c = -2147483648;
    int d = c - 1;
    printf("%d (%x)\n",d,d);

    unsigned int e = 4294967295;
    unsigned int f = e + 1;
    printf("%u (%x)\n",f,f);

    unsigned int g = 0;
    unsigned int h = g - 1;
    printf("%u (%x)\n",h,h);

    return 0;
}
```

• New format specifiers for hexadecimal (%x) and unsigned int (%u)
2.2. char Type

- **Meaning?** Character in a "word"
- **Stored?** 8-bit binary representation using ASCII standard
- **Operations?** Basic integer arithmetic

```c
#include <stdio.h>

int main()
{
    char a = 'e';
    char b = 'c';
    char c = 'e';
    printf("%c%c%c\n", a, b, c);
    return 0;
}
```

- New format specifier for char (%c)
2.3. **float/double Type**

- How can we represent real numbers?
- One option is to use *fixed-point* representation, e.g., 4-bit fixed point with 2-bit integer part and 2-bit fractional part

- Problem with fixed point is it provides a relatively small range; does not enable representing very small nor very large numbers
- An alternative is to use *floating-point* representation, where there is no fixed number of digits before and after the binary point
• C floating-point representation for float/double uses an IEEE standard where each number has three fields: sign bit (s), mantissa (m), and exponent (e)

\[ (-1)^s \times m \times 2^e \]

• Both very small and very large numbers can be represented

<table>
<thead>
<tr>
<th>Number</th>
<th>s</th>
<th>m</th>
<th>e</th>
<th>Floating-Point Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.75</td>
<td>0</td>
<td>1.375</td>
<td>1</td>
<td>((-1)^0 \times 1.375 \times 2^1)</td>
</tr>
<tr>
<td>-2.75</td>
<td>1</td>
<td>1.375</td>
<td>1</td>
<td>((-1)^1 \times 1.375 \times 2^1)</td>
</tr>
<tr>
<td>1536</td>
<td>0</td>
<td>1.5</td>
<td>10</td>
<td>((-1)^0 \times 1.5 \times 2^{10})</td>
</tr>
<tr>
<td>0.1875</td>
<td>0</td>
<td>1.5</td>
<td>-3</td>
<td>((-1)^0 \times 1.5 \times 2^{-3})</td>
</tr>
</tbody>
</table>

• float uses 32 bits and double uses 64 bits to encode real numbers

• Binary encoding of the mantissa and exponent is a little more complex than a straight-forward two’s complement encoding
  
  – The exponent is encoded using a bias, so the stored value is actually \(e + 127\) (e.g., to represent 1, store \(1 + 127 = 128\))
  
  – The integer portion of the mantissa is always assumed to be 1 so we only need to store the fractional portion of the mantissa (e.g., a mantissa of 1.375 is allowed, but a mantissa of 2.75 is not allowed)

1. ```
   float a = 2.75;
   ```
```c
float avg( float x, float y )
{
    float sum = x + y;
    return sum / 2;
}

int main()
{
    float a = 10;
    float b = 15;
    float c = avg( a, b );
    return 0;
}
```

https://www.h-schmidt.net/FloatConverter/IEEE754.html
- There is an infinite number of values between 0 and 1
- float/double have a finite number of bits (precision)
- float is 32-bit and represents **single-precision floating point**
- double is 64-bit and represents **double-precision floating point**
- Limited bits can cause overflow/underflow, but limited precision can also cause other strange behavior

```c
#include <stdio.h>

float avg( float x, float y )
{
    float sum = x + y;
    return sum / 2;
}

int main()
{
    float a = 0.2;
    float b = 0.4;
    float c = avg( a, b );
    printf( "average of %f and %f is %f\n", a, b, c );
    if ( c == 0.3 )
        printf(" the average is 0.3\n" );
    else
        printf(" the average is not 0.3\n" );
    return 0;
}
```

- New format specifier for float/double (**%f**)
- Can also use **%m.nf** with *n* decimal places and *m* minimum width
3. Programmer-Defined Types

In addition to the default types that are included as part of the C programming language (e.g., int, unsigned int, char, float, double), C also enables programmers to define their own new types.

3.1. Typedefs

• A typedef actually does not define a new type
• A typedef simply provides a new alias for an already defined type

```c
typedef type_name new_type_name;
```

• The following code is perfectly fine

```c
typedef unsigned int uint_t;
uint_t a = 2;
uint_t b = 3;
unsigned int c = a + b;
```

3.2. struct Types

• A struct enables bundling multiple variables into a single entity
• A struct definition creates a new type and specifies the type and names of the variables contained within the struct

```c
struct _complex_t {
    double real;
    double imag;
};
typedef struct _complex_t complex_t;
```

• Struct definitions are at global scope just like function definitions
• Struct declaration statement simply creates multiple variables on the stack in a single statement

```c
typedef struct
{
    int x;
    int y;
}
point_t;

point_t point_add( point_t pt1,
                   point_t pt2 )
{
    point_t pt3;
    pt3.x = pt1.x + pt2.x;
    pt3.y = pt1.y + pt2.y;
    return pt3;
}

int main()
{
    point_t pt_a;
    pt_a.x = 2;
    pt_a.y = 3;

    point_t pt_b = { 4, 5 };

    point_t pt_c;
    pt_c = point_add( pt_a, pt_b );

    return 0;
}
```
3.3. *enum* Types

- An *enum* enables creating multiple named constants

```c
#include <stdio.h>

enum color_t {
   COLOR_RED,  
   COLOR_ORANGE,  
   COLOR_YELLOW,  
   COLOR_GREEN,  
   COLOR_BLUE,  
   COLOR_PURPLE
};

void print_color( color_t color )
{
   switch ( color ) {
      case COLOR_RED:  printf("red\n");  break;
      case COLOR_ORANGE:  printf("orange\n");  break;
      case COLOR_YELLOW:  printf("yellow\n");  break;
      case COLOR_GREEN:  printf("green\n");  break;
      case COLOR_BLUE:  printf("blue\n");  break;
      case COLOR_PURPLE:  printf("purple\n");  break;
   }
}

int main()
{
   print_color( COLOR_RED );
   print_color( COLOR_BLUE );
   return 0;
}
```
4. Working With Types

Types can offer strong static guarantees about correctness, but also need to be carefully managed.

4.1. Type Checking

• Compiler will check to ensure types are consistent
• Inconsistent types will cause a compile-time error

```c
typedef struct
{
    int x;
    int y;
} point_t;

point_t point_add( point_t pt1,
                   point_t pt2 )
{
    point_t pt3;
    pt3.x = pt1.x + pt2.x;
    pt3.y = pt1.y + pt2.y;
    return pt3;
}

int main()
{
    int a = 2;
    int b = 3;
    point_t pt_c = point_add( a, b );
    return 0;
}
```

http://cpp.sh/3ecpv
4.2. Type Inference

• Compiler uses type inference to determine type of an expression

1  int   a = 2;
2  int   b = 3;
3  int   c = a + b;  // expr (a + b) has type int
4  int   d = a / b;  // expr (a / b) has type int
5
6  float e = 2.0;
7  float f = 3.0;
8  float g = e + f;  // expr (e + f) has type float
9  float h = e / f;  // expr (e / f) has type float
4.3. Type Conversion

- Compiler uses type conversion if variables have different types
- Compiler must convert types so they match
- Lower precision types can be converted to higher precision types
- Higher precision types can be converted to lower precision types

```c
signed int a = 147483647;
unsigned int b = a;  // no issue

signed int a = -1;
unsigned int b = a;  // careful! b == 4294967295

int a = 2;
float b = a;        // no issue, b == 2.0

float a = 2.5;
double b = a;       // no issue, b == 2.5

float a = 2.5;
int b = a;          // careful!  b == 2

double a = 2.5;
float b = a;        // ok here, but be careful!

int a = 2;
float b = 3;
float c = a + b;    // expr (a + b) has type float
float d = a / b;    // expr (a / b) has type float

unsigned int a = 2;
signed int b = -3;
unsigned int c = a * b;  // expr (e * f) has type signed int
```
• The following example illustrates automatic type conversion

```c
#include <stdio.h>

int avg( int x, int y )
{
    int sum = x + y;
    return sum / 2;
}

int main()
{
    float a = 10;
    float b = 15;
    float c = avg( a, b );
    printf(" average of \%f and \%f is \%f
\", a, b, c );
    return 0;
}
```

http://cpp.sh/927j3
4.4. Type Casting

• Programmers can use type casting to explicitly convert types

```c
#include <stdio.h>

float avg( int x, int y )
{
    int sum = x + y;
    return ((float) sum) / 2.0;
}

int main()
{
    float a = 10;
    float b = 15;
    float c = avg( a, b );
    printf( "average of %f and %f is %f\n", a, b, c );
    return 0;
}
```

• Type of LHS not part of type conversion rules ...
• ... so just specifying a return type of float in avg is not enough
• Could specify the type of sum to be float ...
• ... or use type casting to cast an int into a float