4 Working With Types

4.1. Type Checking .......................................................... 13
4.2. Type Inference .......................................................... 14
4.3. Type Conversion ....................................................... 15
4.4. Type Casting ........................................................... 17

Sections marked with a star (★) are not covered in lecture but are instead covered in the online lecture notes. Students are responsible for all material covered in lecture and in the online lecture notes. Material from the online lecture notes will definitely be assessed in the prelim and final exam.

Copyright © 2018 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.
• In C, the **type** of a variable answers three questions:
  – What is the **meaning** of the variable’s value?
  – How should the variable’s value 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

We will primarily use the following primitive C types

- int: For representing integer numbers
- char: For representing characters
- float and double: For representing real numbers
- const T: For representing constant values of type T
- void: For representing situations where a value is not allowed

2.1. int Type

- Meaning? Integer whole numbers in a limited range
- Stored? 32-bit two’s complement binary representation
- Operations? Basic integer arithmetic

Unlike some productivity-level programming languages, variables of type int cannot represent arbitrarily large or small integers

Such variables have a fixed upper limit and lower limit
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;
}
```
4-Bit 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
- To start, let’s focus on variables of type unsigned int and let’s assume all variables are only four bits

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

```
unsigned int a = 4;
unsigned int b = 15;
unsigned int c = 0;
unsigned int d = a + 1;
unsigned int e = b + 1;
unsigned int f = c - 1;
```
4-Bit Signed Integers

- Now let’s consider variables of type signed int and let’s continue to assume all variables are only four bits.

- There can be multiple ways to encode a given value into a sequence of bits (e.g., sign magnitude, one’s complement, two’s complement).

- The C language specification does not actually specify the exact encoding, but essentially all machines use two’s complement.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Sign</th>
<th>One’s Mag</th>
<th>Two’s Comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>–0</td>
<td>–7</td>
<td>–8</td>
</tr>
<tr>
<td>1001</td>
<td>–1</td>
<td>–6</td>
<td>–7</td>
</tr>
<tr>
<td>1010</td>
<td>–2</td>
<td>–5</td>
<td>–6</td>
</tr>
<tr>
<td>1011</td>
<td>–3</td>
<td>–4</td>
<td>–5</td>
</tr>
<tr>
<td>1100</td>
<td>–4</td>
<td>–3</td>
<td>–4</td>
</tr>
<tr>
<td>1101</td>
<td>–5</td>
<td>–2</td>
<td>–3</td>
</tr>
<tr>
<td>1110</td>
<td>–6</td>
<td>–1</td>
<td>–3</td>
</tr>
<tr>
<td>1111</td>
<td>–7</td>
<td>–0</td>
<td>–1</td>
</tr>
</tbody>
</table>
2. Basic Data Types 2.1. int Type

- 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 + 1 = %d (%x)\n", a, b, b);

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

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

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

    return 0;
}
```

- New format specifiers for hexadecimal (%x) and unsigned int (%u)

Topic 3: C Types
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)
2.4. **const Types**

- **Meaning?** Indicates variable will not change
- **Stored?** Whatever is required for “base” type
- **Operations?** Read-only operations, cannot modify variable

```c
// Constant at global scope
const double PI = 3.1415926535;

int main()
{
    const double a = 2.0;
    int b = a * PI;
    const int d = 15;
    d = b;                  // compile time error!
    return 0;
}
```

2.5. **void Types**

- **Meaning?** No values are allowed
- **Stored?** No storage needed
- **Operations?** None

```c
void print_line( void )
{
    for ( int i = 0; i < 74; i++ )
        printf("-");
}
```

- Technically, we should use `void` for empty parameter lists
- This applies to `main` as well
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

```
typedef type_name new_type_name;
```

- The following code is perfectly fine

```
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

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

[https://repl.it/@cbatten/ece2400-T03-ex3](https://repl.it/@cbatten/ece2400-T03-ex3)
4.2. Type Inference

- Compiler uses **type inference** to determine type of an expression

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

float e = 2.0;
float f = 3.0;
float g = e + f; // expr (e + f) has type float
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( void )
{
    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;
}
```

https://repl.it/@cbatten/ece2400-T03-ex4
4. Working With Types 4.4. Type Casting

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( void )
{
    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`