Introduction to Programming

Week – *5*, Lecture – *3*

Type Ranges and Conversions in C

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

This location, can store some 0s and 1s – just like any other location in the memory

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

This location, can store some 0s and 1s – just like any other location in the memory

For example, the bit pattern `00001010` represents `10`

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

This location, can store some 0s and 1s – just like any other location in the memory

For example, the bit pattern `00001010` represents `10`

- Here, `1010` is the binary equivalent of `10`

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

This location, can store some 0s and 1s – just like any other location in the memory

For example, the bit pattern `00001010` represents `10`

- Here, `1010` is the binary equivalent of `10`
- ... and since the minimum space we can get in the memory is a byte, we've added 4 leading 0s

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

This location, can store some 0s and 1s – just like any other location in the memory

For example, the bit pattern `00001010` represents `10`

- Here, `1010` is the binary equivalent of `10`
- ... and since the minimum space we can get in the memory is a byte, we've added 4 leading 0s

How about negative numbers then?

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

This location, can store some 0s and 1s – just like any other location in the memory

For example, the bit pattern `00001010` represents `10`

- Here, `1010` is the binary equivalent of `10`
- ... and since the minimum space we can get in the memory is a byte, we've added 4 leading 0s

How about negative numbers then?

- There are multiple possible ways to do that

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

This location, can store some 0s and 1s – just like any other location in the memory

For example, the bit pattern `00001010` represents `10`

- Here, `1010` is the binary equivalent of `10`
- ... and since the minimum space we can get in the memory is a byte, we've added 4 leading 0s

How about negative numbers then?

- There are multiple possible ways to do that
- The way it is usually done, is called the *2's complement representation*

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

This location, can store some 0s and 1s – just like any other location in the memory

For example, the bit pattern `00001010` represents `10`

- Here, `1010` is the binary equivalent of `10`
- ... and since the minimum space we can get in the memory is a byte, we've added 4 leading 0s

How about negative numbers then?

- There are multiple possible ways to do that
- The way it is usually done, is called the *2's complement representation*
- We will not discuss it here, because you will study it in a course on Switching Theory

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

This location, can store some 0s and 1s – just like any other location in the memory

For example, the bit pattern `00001010` represents `10`

- Here, `1010` is the binary equivalent of `10`
- ... and since the minimum space we can get in the memory is a byte, we've added 4 leading 0s

How about negative numbers then?

- There are multiple possible ways to do that
- The way it is usually done, is called the *2's complement representation*
- We will not discuss it here, because you will study it in a course on Switching Theory
- However, the easiest of the representation, is called the *sign magnitude representation*

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

This location, can store some 0s and 1s – just like any other location in the memory

For example, the bit pattern `00001010` represents `10`

- Here, `1010` is the binary equivalent of `10`
- ... and since the minimum space we can get in the memory is a byte, we've added 4 leading 0s

How about negative numbers then?

- There are multiple possible ways to do that
- The way it is usually done, is called the *2's complement representation*
- We will not discuss it here, because you will study it in a course on Switching Theory
- However, the easiest of the representation, is called the *sign magnitude representation*
- In this representation, we use the leftmost bit to store the number's sign (0 ➔ positive, 1➔ negative)

The representation of an integer variable

We discussed that a variable, is essentially a name for a location in the memory

This location, can store some 0s and 1s – just like any other location in the memory

For example, the bit pattern `00001010` represents `10`

- Here, `1010` is the binary equivalent of `10`
- ... and since the minimum space we can get in the memory is a byte, we've added 4 leading 0s

How about negative numbers then?

- There are multiple possible ways to do that
- The way it is usually done, is called the *2's complement representation*
- We will not discuss it here, because you will study it in a course on Switching Theory
- However, the easiest of the representation, is called the *sign magnitude representation*
- In this representation, we use the leftmost bit to store the number's sign (0 ➔ positive, 1➔ negative)
- Thus, in sign magnitude representation, `-10` can be represented as `10001010`

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

Mantissa refers to the actual digits in the number (in the binary form)

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

Mantissa refers to the actual digits in the number (in the binary form)

Exponent refers to a power, to which 2 should be raised

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

Mantissa refers to the actual digits in the number (in the binary form)

Exponent refers to a power, to which 2 should be raised

Similar to the integers, the leftmost bit is used to store the sign of the number

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

Mantissa refers to the actual digits in the number (in the binary form)

Exponent refers to a power, to which 2 should be raised

Similar to the integers, the leftmost bit is used to store the sign of the number

For example, `12.5` when converted to binary, becomes `1100.1` or 1.1001×2^{11} (3 ➔ 11)

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

Mantissa refers to the actual digits in the number (in the binary form)

Exponent refers to a power, to which 2 should be raised

Similar to the integers, the leftmost bit is used to store the sign of the number

For example, `12.5` when converted to binary, becomes `1100.1` or 1.1001×2^{11} (3 ➔ 11)

- Remember the scientific notation? This notation is similar to that

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

Mantissa refers to the actual digits in the number (in the binary form)

Exponent refers to a power, to which 2 should be raised

Similar to the integers, the leftmost bit is used to store the sign of the number

For example, `12.5` when converted to binary, becomes `1100.1` or `1.1001 X` 2^{11} (3 ➔ 11)

- Remember the scientific notation? This notation is similar to that
- In scientific notation, you can have any digit from `1` to `9`, on the left of decimal point

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

Mantissa refers to the actual digits in the number (in the binary form)

Exponent refers to a power, to which 2 should be raised

Similar to the integers, the leftmost bit is used to store the sign of the number

For example, `12.5` when converted to binary, becomes `1100.1` or 1.1001×2^{11} (3 ➔ 11)

- Remember the scientific notation? This notation is similar to that
- In scientific notation, you can have any digit from `1` to `9`, on the left of decimal point
- In this notation, the only possible digit on the left of decimal point is `1`

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

Mantissa refers to the actual digits in the number (in the binary form)

Exponent refers to a power, to which 2 should be raised

Similar to the integers, the leftmost bit is used to store the sign of the number

For example, `12.5` when converted to binary, becomes `1100.1` or 1.1001×2^{11} (3 ➔ 11)

- Remember the scientific notation? This notation is similar to that
- In scientific notation, you can have any digit from `1` to `9`, on the left of decimal point
- In this notation, the only possible digit on the left of decimal point is `1`
- … which makes it redundant, and hence, all we need to store is `1001` (the part after decimal)

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

Mantissa refers to the actual digits in the number (in the binary form)

Exponent refers to a power, to which 2 should be raised

Similar to the integers, the leftmost bit is used to store the sign of the number

For example, `12.5` when converted to binary, becomes `1100.1` or `1.1001 X` 2^{11} (3 ➔ 11)

- Remember the scientific notation? This notation is similar to that
- In scientific notation, you can have any digit from `1` to `9`, on the left of decimal point
- In this notation, the only possible digit on the left of decimal point is `1`
- … which makes it redundant, and hence, all we need to store is `1001` (the part after decimal)
- … and `11` (the binary equivalent of 3)

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

Mantissa refers to the actual digits in the number (in the binary form)

Exponent refers to a power, to which 2 should be raised

Similar to the integers, the leftmost bit is used to store the sign of the number

For example, `12.5` when converted to binary, becomes `1100.1` or `1.1001 X` 2^{11} (3 ➔ 11)

- Remember the scientific notation? This notation is similar to that
- In scientific notation, you can have any digit from `1` to `9`, on the left of decimal point
- In this notation, the only possible digit on the left of decimal point is `1`
- … which makes it redundant, and hence, all we need to store is `1001` (the part after decimal)
- … and `11` (the binary equivalent of 3)
- In the above example, `1001` is the mantissa and `11` is the exponent (there will be leading zeros though)

The representation of a real-valued variable

Real numbers are usually represented by a mechanism involving two different components

Mantissa refers to the actual digits in the number (in the binary form)

Exponent refers to a power, to which 2 should be raised

Similar to the integers, the leftmost bit is used to store the sign of the number

For example, `12.5` when converted to binary, becomes `1100.1` or `1.1001 X` 2^{11} (3 ➔ 11)

- Remember the scientific notation? This notation is similar to that
- In scientific notation, you can have any digit from `1` to `9`, on the left of decimal point
- In this notation, the only possible digit on the left of decimal point is `1`
- … which makes it redundant, and hence, all we need to store is `1001` (the part after decimal)
- … and `11` (the binary equivalent of 3)
- In the above example, `1001` is the mantissa and `11` is the exponent (there will be leading zeros though)
- Again, the actual representation may be in 2's complement form (which we will not discuss)

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The possible number of different combinations of 0s and 1s for a fixed number of bits, is also fixed

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The possible number of different combinations of 0s and 1s for a fixed number of bits, is also fixed

- For example, for 2 bits, there are only 4 possible combinations – `00`, `01`, `10` and `11`

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The possible number of different combinations of 0s and 1s for a fixed number of bits, is also fixed

- For example, for 2 bits, there are only 4 possible combinations – `00`, `01`, `10` and `11`

In general, if we have n bits, there are only 2^n different combinations of 0s and 1s

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The possible number of different combinations of 0s and 1s for a fixed number of bits, is also fixed

- For example, for 2 bits, there are only 4 possible combinations – `00`, `01`, `10` and `11`

In general, if we have n bits, there are only 2^n different combinations of 0s and 1s

For example, a character type variable usually takes up 1 byte (or 8 bits) in memory

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The possible number of different combinations of 0s and 1s for a fixed number of bits, is also fixed

- For example, for 2 bits, there are only 4 possible combinations – `00`, `01`, `10` and `11`

In general, if we have n bits, there are only 2^n different combinations of 0s and 1s

For example, a character type variable usually takes up 1 byte (or 8 bits) in memory

- So, there are only $2^8 = 256$ different possibilities

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The possible number of different combinations of 0s and 1s for a fixed number of bits, is also fixed

- For example, for 2 bits, there are only 4 possible combinations – `00`, `01`, `10` and `11`

In general, if we have n bits, there are only 2^n different combinations of 0s and 1s

For example, a character type variable usually takes up 1 byte (or 8 bits) in memory

- So, there are only $2^8 = 256$ different possibilities
- Usually, the representation for a character is its ASCII value in binary (which only requires 7 bits by the way)

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The possible number of different combinations of 0s and 1s for a fixed number of bits, is also fixed

- For example, for 2 bits, there are only 4 possible combinations – `00`, `01`, `10` and `11`

In general, if we have n bits, there are only 2^n different combinations of 0s and 1s

For example, a character type variable usually takes up 1 byte (or 8 bits) in memory

- So, there are only $2^8 = 256$ different possibilities
- Usually, the representation for a character is its ASCII value in binary (which only requires 7 bits by the way)

Similarly, an integer (say a `short`) that occupies 2 bytes, can only have $2^{16} = 65536$ different values

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The possible number of different combinations of 0s and 1s for a fixed number of bits, is also fixed

- For example, for 2 bits, there are only 4 possible combinations – `00`, `01`, `10` and `11`

In general, if we have n bits, there are only 2^n different combinations of 0s and 1s

For example, a character type variable usually takes up 1 byte (or 8 bits) in memory

- So, there are only $2^8 = 256$ different possibilities
- Usually, the representation for a character is its ASCII value in binary (which only requires 7 bits by the way)

Similarly, an integer (say a `short`) that occupies 2 bytes, can only have $2^{16} = 65536$ different values

- Based on the representation, it may mean that the range gets “split” over positive and negative values

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The possible number of different combinations of 0s and 1s for a fixed number of bits, is also fixed

- For example, for 2 bits, there are only 4 possible combinations – `00`, `01`, `10` and `11`

In general, if we have n bits, there are only 2^n different combinations of 0s and 1s

For example, a character type variable usually takes up 1 byte (or 8 bits) in memory

- So, there are only $2^8 = 256$ different possibilities
- Usually, the representation for a character is its ASCII value in binary (which only requires 7 bits by the way)

Similarly, an integer (say a `short`) that occupies 2 bytes, can only have $2^{16} = 65536$ different values

- Based on the representation, it may mean that the range gets “split” over positive and negative values
- Hence, the range for such a variable is $-32,768$ to $32,767$ (if we include 0, these are 65536 numbers)

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The possible number of different combinations of 0s and 1s for a fixed number of bits, is also fixed

- For example, for 2 bits, there are only 4 possible combinations – `00`, `01`, `10` and `11`

In general, if we have n bits, there are only 2^n different combinations of 0s and 1s

For example, a character type variable usually takes up 1 byte (or 8 bits) in memory

- So, there are only $2^8 = 256$ different possibilities
- Usually, the representation for a character is its ASCII value in binary (which only requires 7 bits by the way)

Similarly, an integer (say a `short`) that occupies 2 bytes, can only have $2^{16} = 65536$ different values

- Based on the representation, it may mean that the range gets “split” over positive and negative values
- Hence, the range for such a variable is $-32,768$ to $32,767$ (if we include `0`, these are `65536` numbers)
- Some languages, like C, may allow a variable to be “unsigned”, meaning that their values cannot be negative

The concept of “range” of a variable

Whether the variable is storing integers or real values, they must be converted to a set of 0s and 1s

The possible number of different combinations of 0s and 1s for a fixed number of bits, is also fixed

- For example, for 2 bits, there are only 4 possible combinations – `00`, `01`, `10` and `11`

In general, if we have n bits, there are only 2^n different combinations of 0s and 1s

For example, a character type variable usually takes up 1 byte (or 8 bits) in memory

- So, there are only $2^8 = 256$ different possibilities
- Usually, the representation for a character is its ASCII value in binary (which only requires 7 bits by the way)

Similarly, an integer (say a `short`) that occupies 2 bytes, can only have $2^{16} = 65536$ different values

- Based on the representation, it may mean that the range gets “split” over positive and negative values
- Hence, the range for such a variable is $-32,768$ to $32,767$ (if we include `0`, these are `65536` numbers)
- Some languages, like C, may allow a variable to be “unsigned”, meaning that their values cannot be negative
- In this case, the range for such a variable gets “shifted” – from `0` to `65535` (again, the total is the same)

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

- Different languages have different mechanisms to handle such scenarios

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

- Different languages have different mechanisms to handle such scenarios
- Usually, the mechanism involves *converting* any operands so that all operands have a single type

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

- Different languages have different mechanisms to handle such scenarios
- Usually, the mechanism involves *converting* any operands so that all operands have a single type

This conversion may be *implicit*, i.e. the language does it automatically

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

- Different languages have different mechanisms to handle such scenarios
- Usually, the mechanism involves *converting* any operands so that all operands have a single type

This conversion may be *implicit*, i.e. the language does it automatically

- Implicit conversions usually aim to “promote” types of smaller range to that with a higher range

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

- Different languages have different mechanisms to handle such scenarios
- Usually, the mechanism involves *converting* any operands so that all operands have a single type

This conversion may be *implicit*, i.e. the language does it automatically

- Implicit conversions usually aim to “promote” types of smaller range to that with a higher range

Or, it could be done explicitly, by using mechanisms such as *type casting* or type conversion

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

- Different languages have different mechanisms to handle such scenarios
- Usually, the mechanism involves *converting* any operands so that all operands have a single type

This conversion may be *implicit*, i.e. the language does it automatically

- Implicit conversions usually aim to “promote” types of smaller range to that with a higher range

Or, it could be done explicitly, by using mechanisms such as *type casting* or type conversion

- Type casting involves explicitly changing the type of a value to something else

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

- Different languages have different mechanisms to handle such scenarios
- Usually, the mechanism involves *converting* any operands so that all operands have a single type

This conversion may be *implicit*, i.e. the language does it automatically

- Implicit conversions usually aim to “promote” types of smaller range to that with a higher range

Or, it could be done explicitly, by using mechanisms such as *type casting* or type conversion

- Type casting involves explicitly changing the type of a value to something else
- The actual behaviour for type casting, may vary from one language to other (as well as one type to other)

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

- Different languages have different mechanisms to handle such scenarios
- Usually, the mechanism involves *converting* any operands so that all operands have a single type

This conversion may be *implicit*, i.e. the language does it automatically

- Implicit conversions usually aim to “promote” types of smaller range to that with a higher range

Or, it could be done explicitly, by using mechanisms such as *type casting* or type conversion

- Type casting involves explicitly changing the type of a value to something else
- The actual behaviour for type casting, may vary from one language to other (as well as one type to other)
- For example, in C, if you type cast a real number to an integer, its part after the decimal is simply truncated

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

- Different languages have different mechanisms to handle such scenarios
- Usually, the mechanism involves *converting* any operands so that all operands have a single type

This conversion may be *implicit*, i.e. the language does it automatically

- Implicit conversions usually aim to “promote” types of smaller range to that with a higher range

Or, it could be done explicitly, by using mechanisms such as *type casting* or type conversion

- Type casting involves explicitly changing the type of a value to something else
- The actual behaviour for type casting, may vary from one language to other (as well as one type to other)
- For example, in C, if you type cast a real number to an integer, its part after the decimal is simply truncated

Usually, the compiler tries to perform implicit conversions in a way that there is no loss of precision

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

- Different languages have different mechanisms to handle such scenarios
- Usually, the mechanism involves *converting* any operands so that all operands have a single type

This conversion may be *implicit*, i.e. the language does it automatically

- Implicit conversions usually aim to “promote” types of smaller range to that with a higher range

Or, it could be done explicitly, by using mechanisms such as *type casting* or type conversion

- Type casting involves explicitly changing the type of a value to something else
- The actual behaviour for type casting, may vary from one language to other (as well as one type to other)
- For example, in C, if you type cast a real number to an integer, its part after the decimal is simply truncated

Usually, the compiler tries to perform implicit conversions in a way that there is no loss of precision

- Thus, implicit type casting usually converts a data type of narrower range to that with a higher range

Conversion of types

What happens when two operands of different types are used in an expression or assignment?

- Different languages have different mechanisms to handle such scenarios
- Usually, the mechanism involves *converting* any operands so that all operands have a single type

This conversion may be *implicit*, i.e. the language does it automatically

- Implicit conversions usually aim to “promote” types of smaller range to that with a higher range

Or, it could be done explicitly, by using mechanisms such as *type casting* or type conversion

- Type casting involves explicitly changing the type of a value to something else
- The actual behaviour for type casting, may vary from one language to other (as well as one type to other)
- For example, in C, if you type cast a real number to an integer, its part after the decimal is simply truncated

Usually, the compiler tries to perform implicit conversions in a way that there is no loss of precision

- Thus, implicit type casting usually converts a data type of narrower range to that with a higher range
- There are some exceptions to it – for instance, when a double constant is assigned to a float variable

The `sizeof()` operator in C

A lot of things in C are actually *implementation specific*

The `sizeof()` operator in C

A lot of things in C are actually *implementation specific*

The most common problem that you may face in this regard, are the sizes of different data types

The `sizeof()` operator in C

A lot of things in C are actually *implementation specific*

The most common problem that you may face in this regard, are the sizes of different data types

They may differ from one machine to another

The `sizeof()` operator in C

A lot of things in C are actually *implementation specific*

The most common problem that you may face in this regard, are the sizes of different data types

They may differ from one machine to another

The `sizeof()` operator in C (as well as C++ and Java) can be used to know the same

The `sizeof()` operator in C

A lot of things in C are actually *implementation specific*

The most common problem that you may face in this regard, are the sizes of different data types

They may differ from one machine to another

The `sizeof()` operator in C (as well as C++ and Java) can be used to know the same

- It can be used with type arguments (such as `int` or `double`)…

The `sizeof()` operator in C

A lot of things in C are actually *implementation specific*

The most common problem that you may face in this regard, are the sizes of different data types

They may differ from one machine to another

The `sizeof()` operator in C (as well as C++ and Java) can be used to know the same

- It can be used with type arguments (such as `int` or `double`)…
- … with variables, as well as expressions

The `sizeof()` operator in C

A lot of things in C are actually *implementation specific*

The most common problem that you may face in this regard, are the sizes of different data types

They may differ from one machine to another

The `sizeof()` operator in C (as well as C++ and Java) can be used to know the same

- It can be used with type arguments (such as `int` or `double`)…
- … with variables, as well as expressions

You can use the `sizeof()` operator to find out many implicit conversion rules of your implementation

The `sizeof()` operator in C

```
printf("On this machine,\n");
printf("int is %lu bytes,\n", sizeof(int));
printf("float is %lu bytes,\n", sizeof(float));
printf("double is %lu bytes,\n", sizeof(double));

printf("The result of (i + c) takes %lu bytes\n", sizeof((i + c)));
printf("The result of (i + f) takes %lu bytes\n", sizeof((i + f)));
printf("The result of (i + d) takes %lu bytes\n", sizeof((i + d)));
```

```
int is 4 bytes,
float is 4 bytes,
double is 8 bytes,
The result of (i + c) takes 4 bytes
The result of (i + f) takes 4 bytes
The result of (i + d) takes 8 bytes
```

Homework !!

Read the following articles

- https://www.geeksforgeeks.org/type-conversion-c/
- https://docs.microsoft.com/en-us/cpp/c-language/c-floating-point-constants?view=msvc-160
- https://stackoverflow.com/questions/33163772/what-is-the-difference-between-casting-to-float-and-adding-f-as-a-suffix-whe