C Programming: Deck 3

Type Conversions

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Topics Covered

1. Introduction to Type Conversion
2. Implicit Type Conversion (Widening)
3. Explicit Type Conversion (Casting)
4. Integer Promotion
5. Conversion in Expressions
6. Program Examples
7. Conversion Rules Summary
8. Common Pitfalls
9. Best Practices
10. Practice Exercises

What is Type Conversion?

- Converting a value from one data type to another
- Necessary when mixing different types in expressions
- Can happen automatically or manually
- Important to understand to avoid data loss and bugs

Two Types of Conversion:

1. **Implicit Conversion** (Automatic) - Done by compiler
2. **Explicit Conversion** (Casting) - Done by programmer

Why Do We Need Type Conversion?

- Different types cannot be directly mixed
- Operations require operands of same type
- Function arguments must match parameter types
- Assignment requires compatible types

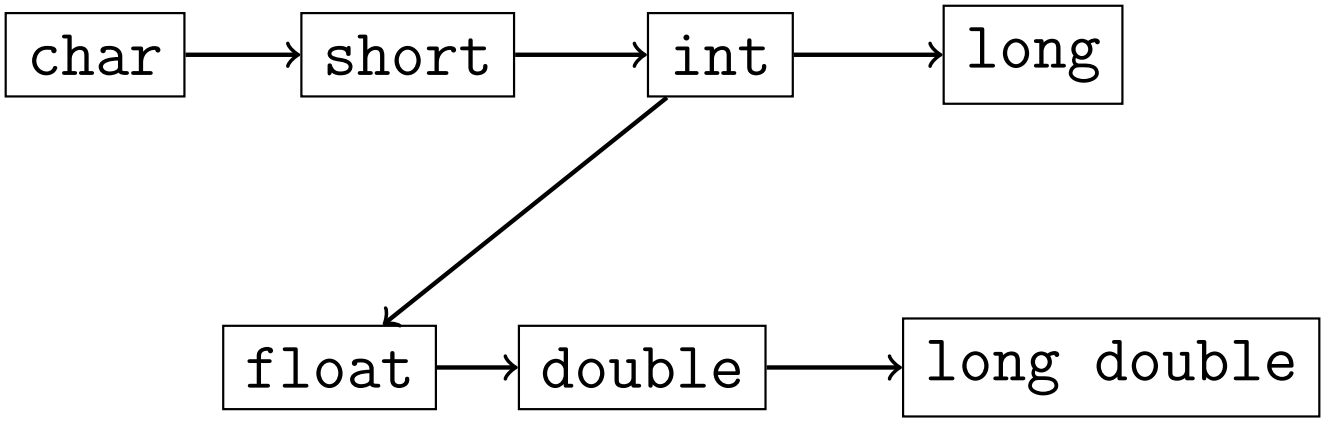
Example Scenarios:

- Adding an `int` and a `float`
- Dividing two integers but want decimal result
- Storing a `double` value in an `int` variable
- Passing arguments to functions

Implicit Conversion - Automatic

- Also called **type promotion** or **coercion**
- Performed automatically by the compiler
- Happens when mixing types in expressions
- Converts smaller type to larger type
- **No data loss** in widening conversions
- Programmer doesn't need to do anything

Widening Conversion Hierarchy



Direction: Smaller $\rightarrow$ Larger (Safe, No Loss)

Rules for Implicit Conversion

When mixing types in an expression:

1. If one operand is `long double`, convert other to `long double`
2. Else if one is `double`, convert other to `double`
3. Else if one is `float`, convert other to `float`
4. Else perform integer promotions:
   - `char`, `short` promoted to `int`
   - If one is `unsigned long`, convert to `unsigned long`
   - If one is `long`, convert to `long`
   - If one is `unsigned`, convert to `unsigned`

Explicit Conversion - Casting

- Manually specified by the programmer
- Uses the cast operator: `(type)`
- Can convert from larger type to smaller type
- May result in data loss
- Gives programmer full control

Syntax:

```
(target_type) expression
```

Examples:

```
(int) 3.14       // Result: 3
(float) 5        // Result: 5.0
(char) 65        // Result: 'A'
```

Narrowing Conversion

- Converting from larger type to smaller type
- Requires explicit casting
- **May cause data loss**
- Decimal part is truncated (not rounded)
- Value may be out of range for target type

Examples of Data Loss:

- `double` to `int`: loses decimal part
- `int` to `char`: may lose high-order bits
- `long` to `short`: may overflow

Integer Promotion Rules

- `char` and `short` are promoted to `int`
- Happens automatically in expressions
- Done before any arithmetic operation
- Ensures operations work on at least `int` size

Why?

- CPU performs arithmetic on register-sized integers
- Most CPUs don't have byte-sized arithmetic operations
- Improves performance and consistency

Arithmetic Conversions

In mixed-type arithmetic expressions:

1. Both operands are promoted to a common type
2. Operation is performed in that type
3. Result has that type

Example: `int + float`

- `int` is converted to `float`
- Addition performed in `float`
- Result is `float`

Assignment Conversions

When assigning to a variable:

- Right-hand side is converted to type of left-hand side
- May involve narrowing (with potential data loss)
- Compiler may warn about narrowing

Examples:

- `int x = 3.7;` $\rightarrow$ `x` becomes 3
- `float y = 5;` $\rightarrow$ `y` becomes 5.0
- `char c = 300;` $\rightarrow$ overflow, undefined result

Program 1: Implicit Widening - int to float

```c
#include <stdio.h>
int main() {
  int i = 10;
  float f;
  f = i;  // Implicit
  printf("int: %d\n", i);
  printf("float: %f\n", f);
  return 0;
}
```

Output:

```
int: 10
float: 10.000000
```

Explanation:

- `int` automatically converted to `float`
- No data loss
- Decimal part added

Program 2: Implicit Widening - char to int

```c
#include <stdio.h>
int main() {
  char c = 'A';
  int i;
  i = c;  // Implicit
  printf("char: %c\n", c);
  printf("int: %d\n", i);
  return 0;
}
```

Output:

```
char: A
int: 65
```

Explanation:

- Character stored as ASCII value
- Widened to `int`
- 'A' = 65

Program 3: Narrowing - float to int (Data Loss)

```c
#include <stdio.h>
int main() {
  float f = 3.14159;
  int i;
  i = f;  // Implicit
  printf("float: %f\n", f);
  printf("int: %d\n", i);
  return 0;
}
```

Output:

```
float: 3.141590
int: 3
```

Explanation:

- Decimal part truncated
- NOT rounded
- Data loss occurs
- Compiler may warn

Program 4: Narrowing - double to int

```c
#include <stdio.h>
int main() {
  double d1 = 9.99;
  double d2 = -5.67;
  int i1, i2;
  i1 = d1;
  i2 = d2;
  printf("%.2lf -> %d\n",
         d1, i1);
  printf("%.2lf -> %d\n",
         d2, i2);
  return 0;
}
```

Output:

```
9.99 -> 9
-5.67 -> -5
```

Explanation:

- Truncation towards zero
- 9.99 becomes 9
- -5.67 becomes -5
- Not rounding!

Program 5: Explicit Casting - float to int

```c
#include <stdio.h>
int main() {
  float f = 7.89;
  int i;
  i = (int)f;  // Explicit
  printf("Original: %f\n",
         f);
  printf("Casted: %d\n", i);
  printf("Float still: %f\n",
         f);
  return 0;
}
```

Output:

```
Original: 7.890000
Casted: 7
Float still: 7.890000
```

Explanation:

- Explicit cast to `int`
- Original value unchanged
- Programmer's intention clear

Program 6: Explicit Casting - int to char

```c
#include <stdio.h>
int main() {
  int i1 = 65;
  int i2 = 97;
  char c1, c2;
  c1 = (char)i1;
  c2 = (char)i2;
  printf("%d -> %c\n",
         i1, c1);
  printf("%d -> %c\n",
         i2, c2);
  return 0;
}
```

Output:

```
65 -> A
97 -> a
```

Explanation:

- ASCII values converted to characters
- 65 = 'A'
- 97 = 'a'
- Explicit cast used

Program 7: Mixed Type Arithmetic

```c
#include <stdio.h>
int main() {
  int i = 5;
  float f = 2.5;
  float result;
  result = i + f;
  printf("int: %d\n", i);
  printf("float: %f\n", f);
  printf("Result: %f\n",
         result);
  return 0;
}
```

Output:

```
int: 5
float: 2.500000
Result: 7.500000
```

Explanation:

- `i` promoted to `float`
- 5 becomes 5.0
- Addition in `float`
- Result is `float`

Program 8: Integer Division vs Float Division

```c
#include <stdio.h>
int main() {
  int a = 7, b = 2;
  int result1;
  float result2;
  result1 = a / b;
  result2 = a / b;
  printf("Int result: %d\n",
         result1);
  printf("Float var: %f\n",
         result2);
  return 0;
}
```

Output:

```
Int result: 3
Float var: 3.000000
```

Explanation:

- Both use integer division
- $7/2 = 3$ (truncated)
- `result2` gets 3.0
- Still data loss!

Program 9: Correct Float Division

```c
#include <stdio.h>
int main() {
  int a = 7, b = 2;
  float result;
  // Cast to get decimal
  result = (float)a / b;
  printf("Division: %d/%d\n",
         a, b);
  printf("Result: %f\n",
         result);
  return 0;
}
```

Output:

```
Division: 7/2
Result: 3.500000
```

Explanation:

- Cast a to `float`
- b promoted to `float`
- Float division performed
- Correct decimal result

Program 10: Average Calculation (Wrong)

```c
#include <stdio.h>
int main() {
  int a = 10, b = 20, c = 25;
  float avg;
  avg = (a + b + c) / 3;
  printf("Numbers: %d, %d, %d\n",
         a, b, c);
  printf("Average: %f\n", avg);
  return 0;
}
```

Output:

```
Numbers: 10, 20, 25
Average: 18.000000
```

Explanation:

- Sum: 55 (int)
- $55 / 3 = 18$ (int division)
- Result: 18.0
- Wrong! Should be 18.333...

Program 11: Average Calculation (Correct)

```c
#include <stdio.h>
int main() {
  int a = 10, b = 20, c = 25;
  float avg;
  avg = (a + b + c) / 3.0;
  printf("Numbers: %d, %d, %d\n",
         a, b, c);
  printf("Average: %f\n", avg);
  return 0;
}
```

Output:

```
Numbers: 10, 20, 25
Average: 18.333333
```

Explanation:

- Divide by 3.0 (double)
- Sum promoted to double
- Double division
- Correct result!

Program 12: char Arithmetic and Promotion

```c
#include <stdio.h>
int main() {
  char c1 = 'A';
  char c2 = 'B';
  int diff;
  diff = c2 - c1;
  printf("c1: %c (%d)\n",
         c1, c1);
  printf("c2: %c (%d)\n",
         c2, c2);
  printf("Difference: %d\n",
         diff);
  return 0;
}
```

Output:

```
c1: A (65)
c2: B (66)
Difference: 1
```

Explanation:

- Both chars promoted to int
- ASCII: B(66) - A(65)
- Result: 1
- Useful for char operations

Program 13: Overflow in Narrowing

```c
#include <stdio.h>
int main() {
  int i = 300;
  char c;
  c = (char)i;
  printf("int value: %d\n", i);
  printf("char value: %d\n",
         c);
  printf("As character: %c\n",
         c);
  return 0;
}
```

Output:

```
int value: 300
char value: 44
As character: ,
```

Explanation:

- char range: -128 to 127
- 300 overflows
- Wraps around: 300 % 256 = 44
- Undefined behavior!

Program 14: Comparing Conversions

```c
#include <stdio.h>
int main() {
  int i = 100;
  float f = 3.14;
  double d = 2.71828;
  char c = 'Z';
  printf("Original values:\n");
  printf("int: %d, float: %.2f, double: %.5lf, char: %c\n\n", i, f, d, c);
  printf("Implicit conversions:\n");
  printf("int to float: %f\n", (float)i);
  printf("float to double: %.5lf\n", (double)f);
  printf("char to int: %d\n", (int)c);
  printf("\nNarrowing conversions:\n");
  printf("double to int: %d\n", (int)d);
  printf("float to int: %d\n", (int)f);
  printf("int to char: %c\n", (char)i);
  return 0;
}
```

Program 14: Output

```
Original values:
int: 100, float: 3.14, double: 2.71828, char: Z

Implicit conversions:
int to float: 100.000000
float to double: 3.14000
char to int: 90

Narrowing conversions:
double to int: 2
float to int: 3
int to char: d
```

Note: 100 in ASCII = 'd', showing potential unexpected results in narrowing.

Program 15: Celsius to Fahrenheit

```c
#include <stdio.h>
int main() {
  float celsius = 25.0;
  float fahrenheit;
  fahrenheit =
    (9.0/5.0) * celsius + 32;
  printf("Celsius: %.2f\n",
         celsius);
  printf("Fahrenheit: %.2f\n",
         fahrenheit);
  return 0;
}
```

Output:

```
Celsius: 25.00
Fahrenheit: 77.00
```

Explanation:

- Use 9.0 and 5.0 for float division
- Otherwise $9/5 = 1$ (integer)
- Correct formula application

Type Conversion Hierarchy

From	To	Result
char	int	Safe, widening
short	int	Safe, widening
int	long	Safe, widening
int	float	Safe, may lose precision
float	double	Safe, widening
double	int	Data loss, truncation
float	int	Data loss, truncation
int	char	May overflow
long	int	May overflow

When to Use Explicit Casting

Use explicit casting when:

1. You want to make your intention clear
2. Converting from larger to smaller type
3. Integer division but need float result
4. Avoiding compiler warnings
5. Working with different numeric types

Benefits:

- Code is more readable
- Intention is explicit
- Avoids unexpected behavior
- Suppresses compiler warnings (when appropriate)

Common Mistakes - Integer Division

Problem:

- Expecting decimal result from integer division

Wrong:

- `float result = 5 / 2;` $\rightarrow$ 2.0 (not 2.5!)

Correct:

- `float result = 5.0 / 2;` $\rightarrow$ 2.5
- `float result = (float)5 / 2;` $\rightarrow$ 2.5
- `float result = 5 / 2.0;` $\rightarrow$ 2.5

Common Mistakes - Truncation

Problem:

- Assuming rounding instead of truncation

Examples:

- `(int)3.7` $\rightarrow$ 3, not 4
- `(int)9.99` $\rightarrow$ 9, not 10
- `(int)-2.5` $\rightarrow$ -2, not -3

Remember:

- Conversion truncates towards zero
- Does NOT round to nearest integer
- Use math functions for rounding

Common Mistakes - Assignment Order

Problem:

- Performing operation before casting

Wrong:

- `float avg = (float)(a + b) / 2;`
- Parentheses cause integer division first
- Then casts result to float

Correct:

- `float avg = (float)(a + b) / 2.0;`
- `float avg = (a + b) / 2.0;`
- Cast one operand or use float literal

Common Mistakes - Overflow

Problem:

- Value too large for target type

Examples:

- `char c = 500;` $\rightarrow$ Overflow
- `short s = 100000;` $\rightarrow$ Overflow

Results:

- Undefined behavior for signed types
- Wraps around for unsigned types
- Always check ranges!

Best Practices for Type Conversion

1. **Be explicit** - Use casts to show intention
2. **Avoid narrowing** - Only when necessary
3. **Check ranges** - Ensure values fit in target type
4. **Use float literals** - Write 3.0 instead of 3 for division
5. **Understand truncation** - Know decimal parts are lost
6. **Watch for overflow** - Especially with small types
7. **Use parentheses** - Make order of operations clear
8. **Test edge cases** - Maximum and minimum values

Summary: Key Points

- **Widening** (small $\rightarrow$ large): Safe, automatic
- **Narrowing** (large $\rightarrow$ small): May lose data, needs casting
- **Integer promotion**: char/short promoted to int
- **Mixed arithmetic**: Smaller type promoted to larger
- **Integer division**: Always truncates (not rounds)
- **Explicit casting**: Use `(type)` syntax
- **Float division**: At least one operand must be float
- **Truncation**: Towards zero, not rounding

Try These!

1. Write a program to divide two integers and get float result
2. Calculate percentage: (marks * 100) / total (avoid integer division)
3. Convert temperature from Fahrenheit to Celsius
4. Demonstrate data loss when converting double to int
5. Show difference between `5/2` and `5.0/2`
6. Write a program showing char promotion in arithmetic

Sample Solution: Integer to Float Division

```c
#include <stdio.h>
int main() {
  int a = 7, b = 3;
  float result;
  result = (float)a / b;
  printf("%d / %d = %f\n",
         a, b, result);
  return 0;
}
```

Output:

```
7 / 3 = 2.333333
```

Sample Solution: Percentage Calculation

```c
#include <stdio.h>
int main() {
  int marks = 85;
  int total = 100;
  float percentage;
  percentage =
    (marks * 100.0) / total;
  printf("Marks: %d/%d\n",
         marks, total);
  printf("Percentage: %.2f%%\n",
         percentage);
  return 0;
}
```

Output:

```
Marks: 85/100
Percentage: 85.00%
```

End of Deck 3

Questions?

Next: Deck 4 - Operators