Short-term Hands-on Supplementary Course on C Programming

Session 1: C Programming Basics

Authors:
Karthik D
Nivedhitha D

May 12, 2022

1 The C Language

- C was created around 1970 to make writing Unix and Unix tools easier.
- Design principles:
  - Small, simple abstractions of hardware
  - Minimalist aesthetic
  - Prioritizes efficiency and minimalism over safety and high-level abstractions
- C is **procedural**, you write functions, rather than define new variable types with classes and call methods on objects
- C is small, fast and efficient.

2 Why C?

- Many tools (and even other languages, like Python!) are built with C making it easier to migrate to several other languages (C++, Java, etc.).
- C is the language of choice for fast, highly efficient programs.
- C is popular for systems programming (operating systems, networking, etc.).
- C lets you work at a lower level to manipulate and understand the underlying system.

3 Programming Language Popularity

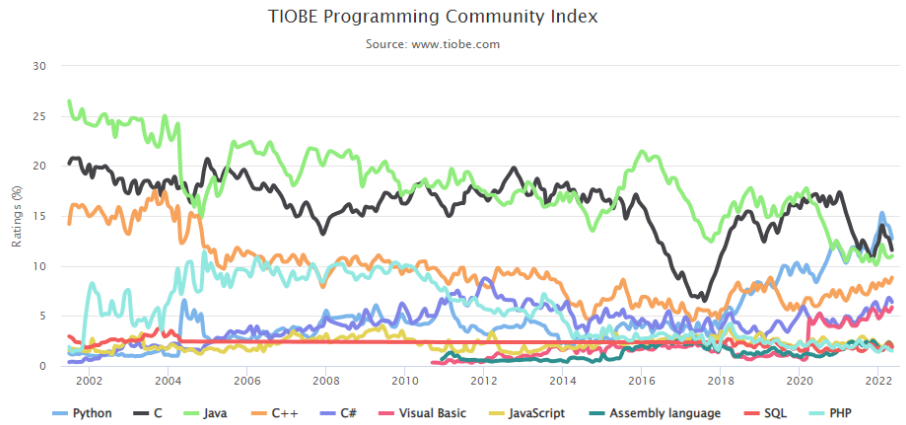

Figure 1: https://www.tiobe.com/tiobe-index/

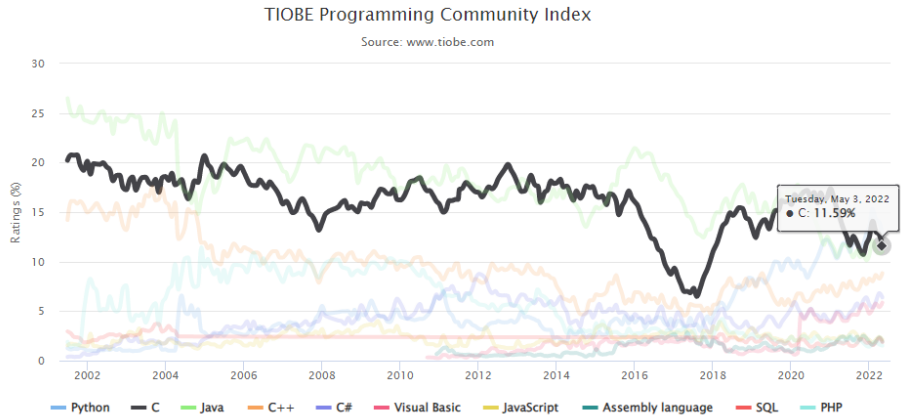

Figure 2: C has been consistently one of the most popular languages since its inception in 1970. It is currently ranked 2nd overall losing out to Python, which is built on C.

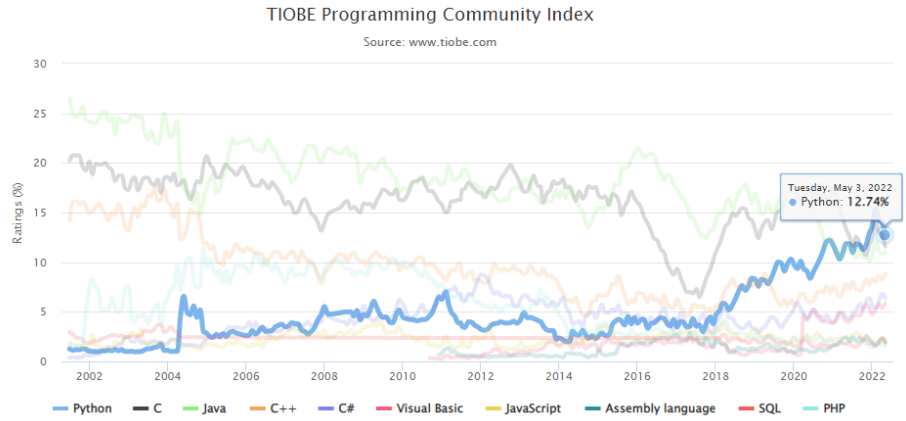

Figure 3: Python is the most popular programming language of 2022.

4 Structure of a C Program

```c
/*
* hello.c
* This program prints a welcome message
* to the user.
*/

// preprocessor directive to include library for printf function
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Listing 1: Our first C program.

- **lines 1-5, 7**: You can include **program comments** for code documentation and improving readability.
  - **lines 1 - 5**: Use **block** style for comments that run for more than a single line.

```c
        /*
        * hello.c
        * This program prints a welcome message
        * to the user.
        */

```

  - **line 7**: Use **inline** style for comments restricted to a single line.

```c
    #include <stdio.h> // for printf

```

- **line 8**: **C** libraries are included using import statements.
  - Standard C libraries are written in angle brackets.

```
1         #include <stdio.h>
2
```

  - Local libraries have quotes:

```
1         #include "lib.h"
2
```

- **lines 10 - 13**: The **main()** function is the entry point for the program. It should always return an integer. Since, C follows UNIX standards, 0 is conventionally used to indicate 'SUCCESS'.

```
1     int main(int argc, char *argv[]) {
2         printf("Hello, world!\n");
3         return 0;
4     }
5
```

- **lines 10**: The main function takes in two command line parameters used to execute the program.
  - **argc**: is the number of arguments in argv.
  - **argv**: is an array of arguments where **char*** is a C string.

```
1     int main(int argc, char *argv[])
2
```

- **lines 11**: The **printf** function prints output to the screen. Each statement of code in C, must end with a **semicolon**.

```
1     printf("Hello, world!\n");
2
```

- **lines 12**: The **main()** function is the entry point for the program. It should always return an integer. Since, C follows UNIX standards, 0 is conventionally used to indicate 'SUCCESS".

```
1     return 0;
2
```

5 Comments

You should use comments generously in the source code to document your C programs. There are two ways to insert a comment in C:

- **Block comments** begin with /* and end with */.
- **Inline** comments begin with // and end with the next newline character.

```
1
2  // Inline comment
3  const double pi = 3.1415926536; // pi is constant
4
5  // Nested comments
6  /* Temporarily removing two lines:
7  const double pi = 3.1415926536; // pi is constant
8  area = pi * r * r // Calculate the area
9  Temporarily removed up to here */
10
11 // Comments don't work within quotations.
12 printf("Comments in C begin with /* or //.\n");
13
14 /*
15 The ellipsis (...) signifies that the open() function has a third,
       optional parameter.
16 The comment explains the usage of the optional third function
       parameter/argument.
17 */
18 int open(const char *name, int mode, ... /* int permissions */);
```

Listing 2: Interesting use cases for comments.

6 Keywords

Keywords are **reserved words** that may not be used as constant or variable or any other identifier names. The following list shows the reserved words in C.

auto	extern	short	while
break	float	signed	_Alignas
case	for	sizeof	_Alignof
char	goto	static	_Atomic
const	if	struct	_Bool
continue	inline	switch	_Complex
default	int	typedef	_Generic
do	long	union	_Imaginary
double	register	unsigned	_Noreturn
else	restrict	void	_Static_assert
enum	return	volatile	_Thread_local

Figure 4: The set of reserved keywords in C [C in a Nutshell].

7 Identifiers

The term identifier refers to the names of variables, functions, macros, structures, and other objects defined in a C program. Identifiers can contain the following characters:

1. Keywords cannot be used as identifiers
2. The letters in the basic character set, a–z and A–Z (identifiers are **case-sensitive**)
3. The underscore character, _ (can be the starting character)
4. The decimal digits 0–9, although the first character of an identifier must not be a digit
5. C does not allow punctuation characters such as @, $, and % within identifiers
6. White space cannot be used (combine words using underscore)

```
int Area;
int area;

a
a_s
d12e
exam_1_final
```

Listing 3: List of valid identifiers.

```
2morrow // cannot start with a number

max Time // white spaces not allowed

box-40 // no special characters allowed, \-

cost_in_$ // no special characters allowed, \$

int // keywords not allowed

two*four // no special characters allowed, *

Joe's // no special characters allowed, '

c++ // no special characters allowed, +
```

Listing 4: List of invalid identifiers.

8 Data Types

Programs have to store and process different kinds of data, such as integers and floating-point numbers, in different ways. To this end, the compiler needs to know what kind of data a given value represents. In C, the term object refers to a location in memory whose contents can represent values. Objects that have names are also called variables. An object's type determines how much space the object occupies in memory, and how its possible values are encoded. For example, the same pattern of bits can represent completely different integers depending on whether the data object is interpreted as signed (that is, either positive or negative) or unsigned (and hence unable to represent negative values).

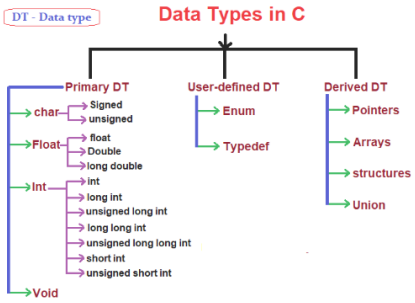

Figure 5: The set of data types in C.

- Basic types
  - Standard and extended integer types
  - Real and complex floating-point types
- Enumerated types
- The type void
- Derived types
  - Pointer types
  - Array types
  - Structure types
  - Union types
  - Function types

```c
char ch = 'A'; // A variable with type char

int iIndex, // Define two int variables and
iLimit = 1000; // initialize the second one.

float height = 1.2345, width = 2.3456; // Float variables have
    single precision.
double area = height * width; // The actual calculation is
    performed with double precision.

double complex z = 1.0 + 2.0 * I;
z *= I; // Rotate z through 90 counterclockwise around the origin
/*
The identifier color is the tag of this enumeration. The
    identifiers in the list black, red, and so on are the
    enumeration constants, and have the type int. Each enumeration
    constant of a given enumerated type represents a certain value,
     which is determined either implicitly by its position in the
    list, or explicitly by initialization with a constant
    expression. A constant without an initialization has the value
    0 if it is the first constant in the list, or the value of the
    preceding constant plus one.
*/
enum color {black, red, green, yellow, blue, white=7, gray}; // 0,
    1, 2, 3, 4, 7, 8

// The type void

void sum (int a, int b); // This function won't return any value to
     the calling function.
int sum (int a, int b); // This function will return value to the
    calling function.
```

Listing 5: Examples of data type declarations.

9 Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type (**char, int, float, double, void**), which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. C programming language also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Structure, Union, etc. For this chapter, let us study only basic variable types.

```c
// Variable definition:
int a, b;
int c; float f;

// Actual initialization
a =10;
b =20;
c = a + b;

// Combined
char grade='A', section = 'B';
```

Listing 6: Variable declaration examples.

10 Constants

Constants refer to **fixed values** that the program may not alter during its execution. These fixed values are also called **literals**. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are enumeration constants as well. Constants are treated just like regular variables except that **their values cannot be modified after their definition**.

```
// Integer Constants
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */

// Floating-point Constants
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */

// Character Constants
'A'
'e'

// String Constants
"hello, dear"
"hello, \
dear"
"hello, dear"

// Escape Sequences
printf("Hello\tWorld\n\n");
```

Listing 7: Constant declaration examples.

10.1 The #define Preprocessor Directive

```
#include <stdio.h>

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main() {
   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

Listing 8: Using the #define preprocessor for defining constants.

10.2 The const keyword

```c
#include <stdio.h>

int main() {
   const int  LENGTH = 10;
   const int  WIDTH = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

Listing 9: Using the const keyword for defining constants.

11 Basic I/O Statements

Input and Output operations are vital for developing user-interactive programs. C supports a multitude of functions to facilitate I/O operations. However, *scanf* and *printf* are the most commonly used functions, and they handle a versatile range of I/O operations by themselves.

11.1 Primer: Format Specifiers

The format specifiers in C Programming facilitate the performance of different input and output operations. Using this type placeholder strategy, the compiler can understand what type of data i.e. integer, character, string, decimal numbers, etc., is to be expected from a particular variable, access it from memory, apply a folding-cast to a suitable character-array representation and display/read it from standard I/O.

Datatype	Format Specifier
int	%d, %i
char	%c
float	%f
double	%lf
short int	%hd
unsigned int	%u
long int	%li
long long int	%lli
unsigned long int	%lu
unsigned long long int	%llu
signed char	%c
unsigned char	%c
long double	%Lf

Figure 6: Format Specifiers in C.

The need for a format specifier comes down to the fact that we can only *"print"* or *"enter into the computer"*, ASCII characters. Hence, format specifiers offer a mapping between each data type and its corresponding string representation — a rather elegant convenience!

11.2 printf() function - Show Output

The *printf()* function is the most used function in the C language. This function is defined in the *stdio.h* header file and is used to show output on the console (standard output).

This function is used to print a simple text sentence or value of any variable which can be of int, char, float, or any other datatype.

```c
#include <stdio.h>

int main() {
    int day = 20;
    int month = 11;
    int year = 2021;
    printf("The date is: %d-%d-%d", day, month, year);
    return 0;
}
```

Listing 10: printf() with multiple data types.

11.3 scanf() function - Take Input

The scanf() function is a functional inversion of *printf*. It can be used to take any type of data as *input from the user*, as long as we ensure a corresponding match between the data type of the reading variable, and its format specifier.

```c
#include <stdio.h>

int main() {
   char gender;
   int age;
   printf("Enter your age and then gender(M, F or O): ");
   scanf("%d %c", &age, &gender);
   printf("You entered: %d and %c", age, gender);
   return 0;
}
```

Listing 11: scanf() with multiple data types.

11.4 Other I/O Functions

While printf() and scanf() cover nearly every possible I/O need a C programmer might require, a whole range of *syntactically convenient* functions, targeted to perform specific I/O operations, do exist in C. Some of the them are listed in this section.

11.4.1 The getchar(), putchar() pair

The *getchar()* function reads a character from the terminal and returns it as an integer. This function reads only a single character at a time.

You can use this method in a loop in case you want to read more than one character.

The *putchar()* function displays the character passed to it on the screen and returns the same character. This function too displays only a single character at a time.

```c
#include <stdio.h>

void main( )
{
    int c;
    printf("Enter a character");
    /*
        Take a character as input and
        store it in variable c
    */
    c = getchar();
    /*
        display the character stored
        in variable c
    */
    putchar(c);
}
```

Listing 12: getchar() and putchar() in C.

11.4.2 The gets(), puts() pair

The *gets()* function reads a line from stdin(standard input) into the buffer pointed to by str pointer, until either a terminating newline or EOF (end of file) occurs.

Note: Unlike *scanf()* that stops reading characters when it encounters a space, *gets()* reads space as a character too.

The *puts()* function writes a string and a trailing newline to stdout.

```c
#include <stdio.h>

void main()
{
    /* character array of length 100 */
    char str[100];
    printf("Enter a string: ");
    gets(str);
    puts(str);
    getch();
    return 0;
}
```

Listing 13: getchar() and putchar() in C.

12 Operators

Operators essentially constitute a special kind of function that takes one or more parameters and produces an output. It is represented using symbols that tell the compiler to perform specific mathematical manipulations.

Operators can be classified broadly based on domain and representational level of manipulation into one of the following:

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operators
- Bitwise operators

12.1 Arithmetic Operators

An arithmetic operator performs straightforward, plain and simple mathematical operations, namely addition, subtraction, multiplication, division etc on numerical values (constants and variables).

The % operator in C is the equivalent of the *mod* operator in math, that evaluated the remainder on division.

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

Figure 7: Arithmetic Operators in C.

12.2 Relational Operators

A relational operator essentially checks the *"relative"*-ness relationship between two operands. It works on an *assertion-observation* principle, where the result of the expression is a boolean truth value. Owing to the lack of boolean data type in C, it returns 1 if the relation is true; and returns value 0, if the relation is false. Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	`5 == 3` is evaluated to 0
>	Greater than	`5 > 3` is evaluated to 1
<	Less than	`5 < 3` is evaluated to 0
!=	Not equal to	`5 != 3` is evaluated to 1
>=	Greater than or equal to	`5 >= 3` is evaluated to 1
<=	Less than or equal to	`5 <= 3` is evaluated to 0

Figure 8: Relational Operators in C.

12.3 Logical Operators

Logical operators help to logically combine multiple expressions, each of which are typically expected to return a boolean true or false (0 or 1, in C), although they do work for expressions that return values other than these. In these cases, the *boolean truth value* of the operands is used in evaluation. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression `((c==5) && (d>5))` equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression `((c==5) || (d>5))` equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression `!(c==5)` equals to 0.

Figure 9: Logical Operators in C.

12.4 Assignment Operators

An assignment operator is used for assigning a value to a variable. While = is the most common one used, C offers a range of convenient operators called *augmented assignment operators* that can perform and operation on, and assign value to the left-hand-side of the expression.

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Figure 10: Assignment Operators in C.

12.5 Bitwise Operators

Bitwise operators are used in C programming to perform bit-level operations — operations on the bit representations of data, rather than on their decimal form, that we usually operate with.

Bit-level operations make processing faster and save power. They commonly used when there is a need to write machine-efficient code. The compiler, and ultimately the ALU, saves significant compute time in these operations.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Figure 11: Bitwise Operators in C.

12.6 The Ternary Conditional Operator — ?:

We use the ternary operator in C to run one code when the condition is true and another code when the condition is false. The syntax works as,

```
testCondition ? expression1 : expression 2;
```

The testCondition is a boolean expression that results in either true or false. If the condition is:

- true - expression1 (before the colon) is executed

- false - expression2 (after the colon) is executed

For example, it could be used like so,

```c
#include <stdio.h>

int main() {
  int age;

  // take input from users
  printf("Enter your age: ");
  scanf("%d", &age);

  // ternary operator to find if a person can vote or not
  (age >= 18) ? printf("You can vote") : printf("You cannot vote");


  return 0;
}
```

Listing 14: Ternary operator in C

13 Expressions

Expressions in C are similar to expressions in mathematics and they follows tools, similar to what mathematical expressions also follow. They are a basic unit of evaluation and each expression has a value. Say, that an expression returns a value of a particular type.

So, an expression can be made up of variables, it can be made up of constants. These are called the atoms or the basic components of an expression. And these atoms and sub-expressions can be combined to produce larger expressions, using operators.

Expressions can be arithmetic, logical and relational.

```c
int z= x+y // arithmetic expression
a>b //relational
a==b // logical
func(a, b) // function call
```

Listing 15: Expression types in C

Likewise, expressions may also be classified based on their constituent atoms:

Constant expressions consist entirely of constant values. So, the expression, 121 + 17 - 110, is a constant expression because each of the terms of the expression is a constant value.

On the contrary, say j were to be declared as an integer variable, the expression, 180 + 2 – j, would not represent a constant expression.

13.1 Rvalue and Lvalue

When it comes to writing assignment statements in C, understanding Lvalue and Rvalue can be useful.

Keeping things informal, *lvalue* either means *"expression which can be placed on the left-hand side of the assignment operator"*, or refers to *"expression which has a memory address"*. It is a *locator value* that can be resolved into a **named**

region of storage. They typically have a *data type* and a *storage class* associated with them. *rvalue*, on the other hand, is simply all other expressions. They *do not* have an associated storage location. They can be resolved at best, to mere values that could be stored somewhere in memory/storage.

In the following code snippet,

```c
#include <stdio.h>

int main(){
    int a;
    a = 10;
    printf("%d", a);
}
```

Listing 16: Understanding lvalue and rvalue in expressions.

variable *a* is an *lvalue* because it has the valid type *int* and also a storage class *auto* associated with it (this is assigned implicitly here).

13.2 Precedence and Associativity for Evaluation

Naturally, expressions may be composed of multiple classes of operators. When different operators occur together, a resolution strategy becomes imminent to evaluate the expression. That is exactly what operator precedence defines (refer to Figure 13), and achieves.

Operator	*Description*	*Associativity*
() **[]** **.** **->** **++ --**	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- **+ -** **! ~** **(type)** ***** **&** **sizeof**	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
*** / %**	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= **> >=**	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to and not equal to	left to right
&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= **+= -=** ***= /=** **%= &=** **^= |=** **<<= >>=**	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	Comma operator	left to right

Figure 12: Operator Precedence in C.

While precedence slots different operators into a hierarchy, *associativity* resolves sub-expressions that have the same operators of the same precedence.

For instance, 1 == 2 != 3 is resolved *left-to-right*. So the evaluation order can be represented as ((1==2)!=3).

It is highly recommended to use parentheses — ( ) — generously, to avoid ambiguities or side effects associated with expression evaluation.

14 Type Conversion

In practice, expressions in programming languages are seldom *homogeneous*. They contain operands from a mix of data types. Consequently, data types

need to be interconverted, to make logical sense of the expression itself, and to make the evaluation possible in the first place!

The C compiler handles some of the most obvious conversions, from the vast pool of possible combinations.

Type promotion is the go-to strategy in these conversions, when done implicitly. However, some scenarios may present where *type demotion* is the only possibility. Figure **??** represents a hierarchy of data types in C. This structure is laid out based on the representational capacity of data types. For instance, *float* ranks higher on this chart than *int*, because the domain of float — the whole set of numbers it can represent — is much larger and precise.

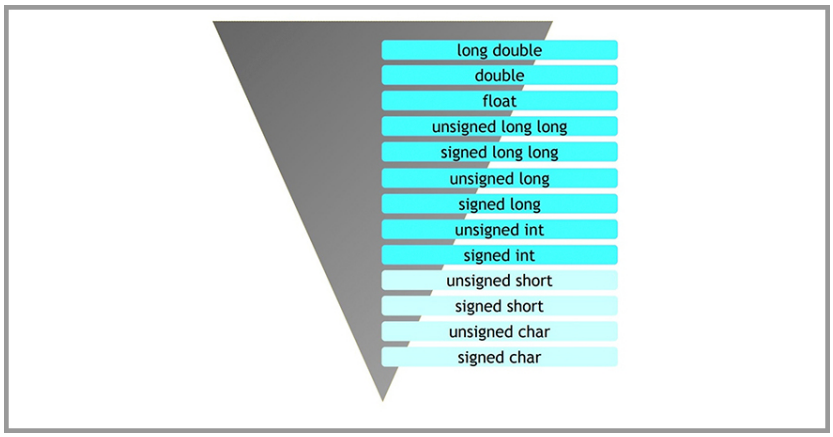

Figure 13: Operator Precedence in C.

Defined concretely, *type promotion* converts from one of the lower ranks to one of the higher ones, while *type demotion* does the opposite. Consequently, type promotion preserves data, while type demotion, may lead to loss of data!

14.1 Type Promotion

When variables are promoted to higher precision, data is preserved. This is automatically done by the compiler for mixed data type expressions.

```
int i;
float f;
f = i +3.14159;  /* i is promoted to float, f =(float)i+3.14159 */
```

Listing 17: Understanding lvalue and rvalue in expressions.

Another conversion done automatically by the compiler is *char* to *int*. This allows comparisons as well as manipulations of character variables.

```
isupper = ( c>='A' && c<='Z' ) ? 1 : 0 ; /* c and literal constants
    are converted to int */
if(!isupper)
c = c - 'a' + 'A' ; /* Subtraction is possible only because of this
    conversion */
```

Listing 18: Character and Integer type conversions

As a rule (with exceptions), the compiler promotes each term in a binary expression to the highest precision operand.

14.2 Type Demotion

```c
#include<stdio.h>

  int main()
  {
      int i_var1, i_var2;
      char c_var1, c_var2;

      float f_var1, f_var2;
      i_var1 = 10;
      c_var1 = 'A';
      f_var1 = 100.92;
      c_var1 = i_var1; // Demotion: variable i_var1 of type int is converted into type char
      i_var1 = f_var1; // Demotion: variable f_var1 of type float is converted into type integer

      printf("Value stored in c_var1 = %dnValue stored in i_var1 = %dn",c_var1,i_var1);
      i_var2 = 20;
      c_var2 = 'B';
      f_var2 = 101.72;
      f_var2 = i_var2; // Promotion: variable i_var2 of type int is converted into type float
      i_var2 = c_var2; // Promotion: variable c_var2 of type char is converted into type int
      printf("Value stored in i_var2 = %dnValue stored in f_var2 = %fn",c_var2,f_var2);
  }
```

Listing 19: Demotion conversions in C

In the given program snippet, integer variable *i_var1* and float variable *f_var1* are demoted in **line 12** and **line 13**, whereas in **line 20** and **line 21**, integer variable *i_var2* and character variable *c_var2* are promoted to type *float* and *int* respectively.

In practice, such type conversions may lead to several consequences that may require particular attention. Some of the could be:

- Some high order bits may be dropped when long is converted to int, or int is converted to short int or char.
- Fractional part may be truncated during conversion of float type to int type.
- When double type is converted to float type, digits are rounded off.
- When a signed type is changed to unsigned type, the sign may be dropped.
- When an int is converted to float, or float to double there will be no increase in accuracy or precision.

15 TUTORIAL 1: Expressions

15.1 Problem 1

Write a program to calculate the shake probability of a Pokeball. Here, a is the catch rate and needs to be read in as the user input. a can be any integral value

in the range [1-255]. This equation is given by:

$$b = \lfloor \frac{65536}{\sqrt{\sqrt{\frac{255}{a}}}} \rfloor$$

15.2 Problem 2

Evaluate the following expressions in order and determine the final output.

```
1
2 int a=5;
3 int b=a++;
4
5 printf("%d %d %d", --a, b--, ++b);
```

15.3 Problem 3

What will be the output of the following code snippet?

```
1 #include<stdio.h>
2 int main()
3 {
4   int a = 555;
5   a = -10, -11, -12;
6   printf("%d", a);
7   return 0;
8 }
```

15.4 Problem 4a

Write a program to swap the values of two variables using a third variable.

15.5 Problem 4b

Write a program to swap the values of two variables without using a third variable.

15.6 Problem 5

Predict the output of the following code snippet.

```
1 #include <stdio.h>
2
3 int main() {
4     // Write C code here
5     int a = 2;
6     int val;
7     val = a == 0 ? 1:
8           a == 1 ? 2:
9           4;
10    printf("%d", val);
11    return 0;
12 }
```

15.7 Problem 6

Predict the output of the following code snippet.

```c
#include <stdio.h>
int main(void)
{
  //variables
  int x, a, b;

  //initialise
  a = 4;
  b = 2;

  //compute x
  x = a * a - 3 * b + a / b;

  //output
  printf("x = %d\n", x);

  printf("End of code\n");
  return 0;
}
```