Computer Programming & Problem Solving

CS100

Mrs Sanga G. Chaki
Department of Computer Science and Engineering
National Institute of Technology, Goa
December, 2022

Topics - Miscellaneous

1. **Operator hierarchy**
2. **Operator associativity**
3. **Goto statements**
4. **Build process**
5. **Variadic Functions**

Operator Hierarchy and Associativity

Operator Hierarchy in C

1. While executing an arithmetic statement, which has two or more operators, how is the sequence of execution decided?
2. Example: i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8

Operator Hierarchy in C

1. All the arithmetic expressions get evaluated by using two properties of operators. They are,
   a) Hierarchy of operators
   b) Associativity of operators
2. The priority or precedence in which the operations in an arithmetic statement are performed is called the hierarchy of operators.

Operator Associativity in C

1. When an expression contains two operators of an equal priority then the tie between them is resolved by using the associativity of operators.

2. Two types of associativity in C

    a) Left to Right Associativity - the left operand must be unambiguous
    b) Right to Left Associativity - the right operand must be unambiguous

Operator Associativity in C

1. Left to Right Associativity:
2. Let's consider an expression: **Result = 11 / 5 * 4**
3. There is a tie between the operators having the same precedence, that is between (/) and (*). This tie gets resolved by using the associativity of (/) and (*).
4. Both the operators have Left to Right associativity in C.
5. But left operand of / is unambiguous.
6. Therefore firstly (/) operation is performed followed by (*).
7. Result = [11 / 5 * 4] = [2 * 4] = [8]

Operator Associativity in C

1. Right to Left Associativity:
2. Let's consider an expression: **Result = Total = 11**
3. Here both the assignment operators have the same precedence. Therefore, the order of evaluation is decided using the associativity of operators.
4. The assignment operator (=) associates the value from Right to Left.
5. Right operand of second = is unambiguous.
6. Therefore, the second assignment is performed earlier than the first assignment.

Operator Hierarchy in C

1. priority of parentheses ( ) operator is the highest.

2. Example: i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8

i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8	operation: *
i = 1 + 4 / 4 + 8 - 2 + 5 / 8	operation: /
i = 1 + 1+ 8 - 2 + 5 / 8	operation: /
i = 1 + 1 + 8 - 2 + 0	operation: /
i = 2 + 8 - 2 + 0	operation: +
i = 10 - 2 + 0	operation: +
i = 8 + 0	operation : -
i = 8	operation: +

Goto Statements

Goto statements

1. Unconditional jump statement
2. The goto statement in C is used to jump from one block to another block during execution and transfer the flow of execution of the code.
3. Do not use goto in any of your code – it is not considered a good practice.
4. They obscure the flow of control

Goto statements – Example 1

```
1  #include <stdio.h>
2  void checkEvenOrNot(int num){
3      if (num % 2 == 0)
4          goto even;
5      else
6          goto odd;
7  even:
8      printf("%d is even", num);
9      return;
10 odd:
11     printf("%d is odd", num);
12 }
13 int main() {
14     int num = 26;
15     checkEvenOrNot(num);
16     return 0;
17 }
```

Goto Labels

1. Program to check if a number is even or not and print accordingly using the goto statement.

Goto statements – Example 2

1. Program to print numbers from 1 to 10. Which one will work?

```c
#include <stdio.h>
void print(){
    int n = 1;
label:
    printf("%d ",n);
    n++;
    if (n <= 10)
        goto label;
}
int main() {
    print();
    return 0;
}
```

```c
#include <stdio.h>
void print(){
    int n = 1;
    if (n <= 10)
        goto label;
label:
    printf("%d ",n);
    n++;
}
int main() {
    print();
    return 0;
}
```

Build Process

Build Process – what happens when the code *runs*?

1. When the C code runs there is four stages of C code building process
2. Each stage utilizes different ‘tools’ such as
    a) a preprocessor,
    b) compiler,
    c) assembler, and
    d) linker.

Build Process – what happens when the code *runs*?

1. **Preprocessor:**
    a) **All the preprocessor directives are evaluated and replaced.**
    b) **The input file for this stage is *filename.c* file.**
    c) **The output file is *filename.i* or preprocessed file.**
    d) **Strips out comments from the input c file**

Build Process – what happens when the code *runs*?

2. Compiler:

a) **C code gets converted into architecture specific assembly code**

b) **Decomposition of C operations into numerous assembly operations. Each operation itself is a very basic task. Lexical, syntactical, semantic analysis.**

c) **The input file for this stage is filename.i file.**

d) **The output file is filename.s or filename.asm file.**

Build Process – what happens when the code *runs*?

3. Assembler:

a) **Assembly code that is generated by the compiler gets converted into object code by the assembler.**

b) **The input file for this stage is filename.asm file.**

c) **The output file is filename.o or filename.obj file**

d) **Object Code: machine code, with information that allows a linker to see what symbols are in it and symbols it requires in order to work.**

Build Process – what happens when the code *runs*?

4. Linker:

a) **It takes one or more object files as input and combines them to produce a single (usually executable) file.**

b) **Executable file = binary form.**

c) **In this process filename.exe gets made from filename.obj.**

Variadic Functions

Variadic Functions

1. In mathematics and in computer programming, a variadic function is a function which accepts a variable number of arguments.
2. Example: printf(), scanf()
3. It takes one fixed argument and then any number of arguments can be passed.
4. The variadic function consists of at least one fixed variable and then an ellipsis(…) as the last parameter.

Variadic Functions

1. General Syntax:
    a) int function_name(data_type variable_name, ...);
2. Values of the passed arguments can be accessed through a header file: stdarg.h

Variadic Functions

1. Methods:
    a) va_start: enables access to variadic function arguments
    b) va_arg: accesses the next variadic function argument
    c) va_copy: makes a copy of the variadic function arguments
    d) va_end: ends traversal of the variadic function arguments
2. va_list:
    a) holds the information needed by va_start, va_arg, va_end, and va_copy
    b) will be the pointer to the last fixed argument in the variadic function

Variadic Functions: Example

```c
#include <stdarg.h>
#include <stdio.h>
int add(int n, ...){ // Variadic function to add numbers
    int sum = 0;
    va_list ptr; // Declaring pointer to the argument list
    va_start(ptr, n);
    for (int i = 0; i < n; i++){
        // Accessing current variable and pointing to next one
        sum += va_arg(ptr, int);}
    va_end(ptr); // Ending argument list traversal
    return sum;}
int main(){
    printf("\n 1 + 2 = %d",add(2, 1, 2)); // Calling variadic
        function
    printf("\n30 + 40 + 50 = %d",add(3, 30, 40, 50));
    printf("\n6 + 70 + 800 + 9000 = %d",add(4, 6, 70, 800, 9000
        ));
    return 0;}
```

Variadic Functions: Example Output

```
1 + 2 = 3
30 + 40 + 50 = 120
6 + 70 + 800 + 9000 = 9876
```