# **CS 410 C++ to Assembly With Loops Activity Template**

**Step 1:** Explain the functionality of the C++ code.

## C++ Code Functionality

| **C++ Line of Code** | **Explanation of Functionality** |
| --- | --- |
| #include <iostream> | Includes the input/output stream library for input/output functionality |
| using namespace std; | Makes use of the standard namespace |
| int main() | Declares a function named main, the entry point of the program. |
| { | An opening brace to indicate where the logic inside the main function begins |
| int num, i; | Declaration of the variable “num” and “i” of data type integer. |
| int product=1; | Declaration of the variable “product” of data type integer initialized to 1. |
| cout << “Enter a number:\n”; | Outputs to the console “Enter a number:” followed by a new line. Unlike with the use of endl, “\n” does not clear the output buffer, however, it does provide a useful way of keeping output formatted neatly. |
| cin >> num; | cin stands for “character input” and is used to read input from the keyboard. Here, input is taken from the user and the value is stored in the variable num. |
| for(i = num; i>0; i--) | A for loop is a control-flow statement that executes a block of code for a specified number of times. A for loop requires three components that specify how the for loop should execute. The first is the initialization of the for loop with a control variable, the second component is the condition to be checked after each iteration of the for loop, and the third component is how the control variable should be updated. Here we set the control variable “i” to be equal to the variable “num” that was previously taken from user input. Then, we tell the for loop that it should continue to execute or iterate over the loop while the variable “i” is greater than zero. Lastly after each iteration of the for loop, the variable “i” is decremented by one. Therefore once the variable “i” equals 1, the loop will be exited. |
| product = product \* i; | For each iteration of the for loop, the variable “product” is multiplied by the value of the variable “i” and stored back into the variable “product” |
| cout << “The factorial for “ << num << “is: “ <<  product << endl; | Outputs to the console the characters “The factorial for “ followed by the variable num, followed by the characters “is: “ then the variable product, followed by a new line, and because we are using endl here, the output buffer is also cleared.. |
| return 0; | Returns 0 because the main function is of type integer and expects an integer value to be returned. Returning a 0 value indicates to the operating system that the program exited without any errors. |
| } | A closing brace to indicate where the logic inside the main function ends. |

**Step 2:** Convert the C++ file into assembly code.

**Step 3:** Align each line of C++ code with the corresponding blocks of assembly code.

## C++ to Assembly Alignment

| **C++ Line of Code** | **Blocks of Assembly Code** |
| --- | --- |
| #include <iostream> | .type \_ZStL19piecewise\_construct, @object    .size \_ZStL19piecewise\_construct, 1  \_ZStL19piecewise\_construct:    .zero 1    .local  \_ZStL8\_\_ioinit    .comm \_ZStL8\_\_ioinit,1,1 |
| using namespace std; | none |
| int main() | .text  .globl main  .type main, @function |
| { | main:  .LFB1493:    .cfi\_startproc    pushq %rbp    .cfi\_def\_cfa\_offset 16    .cfi\_offset 6, -16    movq  %rsp, %rbp    .cfi\_def\_cfa\_register 6 |
| int num, i; | subq  $32, %rsp |
| int product = 1; | movl  $1, -12(%rbp) |
| cout << “Enter a number:\n”; | leaq .LC0(%rip), %rsi  leaq \_ZSt4cout(%rip), %rdi  call  \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc  @PLT |
| cin >> num; | leaq  -20(%rbp), %rax  movq  %rax, %rsi  leaq  \_ZSt3cin(%rip), %rdi  call  \_ZNSirsERi@PLT |
| for (i = num; i>0; i--)  product = product \* I; | movl  -20(%rbp), %eax  movl  %eax, -16(%rbp)  .L3:  cmpl  $0, -16(%rbp)  jle .L2  movl  -12(%rbp), %eax  imull -16(%rbp), %eax  movl  %eax, -12(%rbp)  subl  $1, -16(%rbp)  jmp .L3 |
|  |  |
| cout << “The factorial for “ << num << “is: “ << product << endl; | .L2:  leaq  .LC1(%rip), %rsi  leaq  \_ZSt4cout(%rip), %rdi  call  \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc  @PLT  movq  %rax, %rdx  movl  -20(%rbp), %eax  movl  %eax, %esi  movq  %rdx, %rdi  call  \_ZNSolsEi@PLT  leaq  .LC2(%rip), %rsi  movq  %rax, %rdi  call  \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc  @PLT  movq  %rax, %rdx  movl  -12(%rbp), %eax  movl  %eax, %esi  movq  %rdx, %rdi  call  \_ZNSolsEi@PLT |
| return 0; | leave  ret |
| } |  |

**Step 4:** Explain how the blocks of assembly code perform the same tasks as the C++ code.

## Assembly Functionality

| **Blocks of Assembly Code** | **Explanation of Functionality** |
| --- | --- |
| // #include <iostream>   1. .type \_ZStL19piecewise\_construct, @object 2. .size \_ZStL19piecewise\_construct, 1 3. \_ZStL19piecewise\_construct: 4. .zero 1 5. .local  \_ZStL8\_\_ioinit 6. .comm \_ZStL8\_\_ioinit,1,1 | (1-4) Declares \_ZStL19piecewise\_construct as an object with a size of 1 byte and initializes it to zero. This object contains all of the source code from the iostream header file  (5-6) Declares \_ZStL8\_\_ioinit as a local symbol and allocates 1 byte for it. |
| // int main()   1. .text 2. .globl main 3. .type main, @function | 1. This tells the assembler that the following code is part of the text/code segment. Executable instructions for the program are stored here. 2. Declares main as global. 3. Indicates that main’s type is a function |
| // {   1. main: 2. .LFB1493: 3. .cfi\_startproc 4. pushq %rbp 5. .cfi\_def\_cfa\_offset 16 6. .cfi\_offset 6, -16 7. movq  %rsp, %rbp 8. .cfi\_def\_cfa\_register 6 | 1. This is a label indicating the start of the main function 2. This is a local label used by the assembler for internal purposes 3. This marks the beginning of the function. 4. Pushes the current value of the base pointer register (%rbp) onto the stack. (Saves the base pointer of the previous stack frame). 5. Offsets CFA 16 bytes from the current stack pointer 6. Specifies that the %rbp register is located at an offset of -16 from the CFA 7. Moves the stack pointer to the base pointer, setting up the new stack frame (sets the base pointer to the current stack pointer). 8. This changes the CFA register to %rbp (register number 6) |
| // int num, i  // all variables are allocated space on the  // stack frame   1. subq  $32, %rsp | 1. Allocates 32 bytes on the stack for local variables. The variable num is stored at -20(%rbp) 20 bytes above the base stack, i is stored at -16(%rbp) 16 bytes above the base stack, and product is stored at -12(%rbp) 12 bytes above the base stack. |
| // int product = 1;   1. movl  $1, -12(%rbp) | 1. movl is the instruction to move a 32-bit value. $1 is the immediate value (constant) to be moved, which is 1. -12(%rbp) is the destination address where the value 1 will be stored. (%rbp) is the base pointer register, which points to the base of the current stack frame, and -12(%rbp) means that we are accessing the memory location that is 12 bytes above the address contained in the %rbp register. |
| // cout << “Enter a number:\n”;   1. leaq .LC0(%rip), %rsi 2. leaq \_ZSt4cout(%rip), %rdi 3. call  \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc   @PLT | 1. Loads the effective address for the first string: .LC0: .string “Enter a number:\n” and is used to compute the address of the string relative to the instruction pointer register (%rip) and stores the computed address in the source index register (%rsi). 2. Loads the effective address for the global object cout relative to the instruction pointer register (%rip) and stores the computed address in the destination register (%rdi). 3. This instruction calls the function to output the string to cout. |
| // cin >> num;   1. leaq  -20(%rbp), %rax 2. movq  %rax, %rsi 3. leaq  \_ZSt3cin(%rip), %rdi 4. call  \_ZNSirsERi@PLT | 1. This instruction loads the effective address of the variable num, -20(%rbp) which is stored 20 bytes above the base pointer register (-20 bytes offset), and stores it into the %rax register. 2. This moves the value in the %rax register to the %rsi register. 3. This loads the effective address of cin relative to the instruction pointer and stores the calculated address of the input stream (cin) in the %rdi register 4. This calls the function to read input from cin into the memory location pointed to by the %rsi register, which is the variable num. |
| // for (i = num; i>0; i--)   1. movl  -20(%rbp), %eax 2. movl  %eax, -16(%rbp) 3. .L3: 4. cmpl  $0, -16(%rbp) 5. jle .L2   // ( lines 6 – 8) are inside the for loop   1. movl  -12(%rbp), %eax   // product = product \* I;   1. imull -16(%rbp), %eax 2. movl  %eax, -12(%rbp) 3. subl  $1, -16(%rbp) 4. jmp .L3 | 1. This moves the value stored at -20(%rbp), which is the variable num, into the %eax register to be temporarily stored for use. 2. This moves the value that was previously stored in the %eax register to the location -16(%rbp) which is -16 offset or 16 bytes above the base pointer. This value is used for the variable i. 3. This is used to indicate the start of the for loop. 4. This compares the loop variable that is stored at -16(%rbp), the variable i, with the value 0. 5. If the loop variable, i, is less than or equal to 0, this instruction will jump to the end of the loop (exits the loop). .L2 is the next portion of the assembly code after the code for the for loop. 6. If the for loop is not exited the code inside the for loop is executed. This line moves the value at -12(%rbp), which is the variable product stored at a -12 offset or 12 bytes above the base pointer register, and temporarily stores its value into the %eax register for modifying this variable in the next multiplication line of code. 7. imull multiplies the value stored at -16(%rbp) (16 bytes above the base pointer register), which is the variable I, by the value stored in the %eax register (the value of product) and then stores this value back into the %eax register. 8. The value in the %eax register is then moved back to the location for the variable product which is -12(%rbp) an -12 offset or 12 bytes above the base pointer register. 9. The value 1 is subtracted from the value stored in -16(%rbp) an -16 offset or 16 bytes above the base pointer register, which is the value of the variable i. This is decrementing the loop variable. 10. This line says to jump back to .L3, which is the beginning of the for loop where the next comparison will then be made to determine if the for loop should be exited. |
| // cout << “The factorial for “ << num << // “is: “ << product << endl;   1. .L2: 2. leaq  .LC1(%rip), %rsi 3. leaq  \_ZSt4cout(%rip), %rdi   // cout << “The factorial for “   1. call  \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc   @PLT   1. movq  %rax, %rdx 2. movl  -20(%rbp), %eax 3. movl  %eax, %esi 4. movq  %rdx, %rdi   // << num   1. call  \_ZNSolsEi@PLT 2. leaq  .LC2(%rip), %rsi 3. movq  %rax, %rdi   // << “is: “   1. call  \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc   @PLT   1. movq  %rax, %rdx 2. movl  -12(%rbp), %eax 3. movl  %eax, %esi 4. movq  %rdx, %rdi   // <<product << endl;   1. call  \_ZNSolsEi@PLT | 1. This line indicates the next portion of code within the main function after the for loop. 2. Loads the effective address for the next string: .LC1: .string “The factorial for ” and is used to compute the address of the string relative to the instruction pointer register (%rip) and stores the computed address in the source index register (%rsi). 3. Loads the effective address for the global object cout relative to the instruction pointer register (%rip) and stores the computed address in the destination register (%rdi). 4. This instruction calls the function to output the string to cout. 5. This instruction moves the result of the previous call from the %rax register to the %rdx register to be stored for future use. 6. This instruction moves the value that is stored at -20(%rbp), an -20 offset or 20 bytes above the base pointer register, which is the variable num, to the %eax register for temporary storage. 7. The value stored in the %eax register (num) is moved to the %esi register to be used for output. 8. Moves the previously stored result from the previous output call (%rdx) into the %rdi register for the next portion of output. 9. Calls the function to print the variable num to cout. 10. Loads the effective address for the next string: .LC2: .string “is: \n ” and is used to compute the address of the string relative to the instruction pointer register (%rip) and stores the computed address in the source index register (%rsi). 11. This instruction moves the result of the previous call from the %rax register to the %rdi register to be stored for future use. 12. Calls the function to print the next part of the string to cout. 13. This instruction moves the result of the previous call from the %rax register to the %rdx register to be stored for future use.   (continued on next page…) |
|  | 1. This moves the value at -12(%rbp), an -12 offset or 12 bytes above the base pointer register, which is the value of the variable product, into the %eax register to be temporarily stored for future use. 2. The value stored in the %eax register (product) is moved to the %esi register to be used for output. 3. This moves the value previously stored in the %rdx register, which is the result of the previous call to cout into the %rdi register to be stored for future use ( the next call to output). 4. Calls the function to print the product variable to cout (output). The endl also clears the output buffer. |
| // return 0;   1. leave 2. ret | 1. Cleans up the stack frame 2. Returns from the function. Returning 0 indicates to the operating system that the program exited without any errors. |