Tutorial Link https://codequotient.com/tutorials/C++ : Inline Functions-1/5b4a01613714c2304e8557ea

TUTORIAL

C++ : Inline Functions-1

Chapter

Function call has its overhead (handling the argument, managing function stack, branch and return). For simple and short functions, you may use inline functions to remove the function call overhead. The keyword inline (before the function's return-type) suggest to the compiler to "expand" the function code in place, instead of performing a function call. For example,

C++

```cpp
#include <iostream>
using namespace std;

inline int max(int n1, int n2)
{
  return (n1 > n2) ? n1 : n2;
}

int main()
{
  int i1 = 5, i2 = 6;
  cout << max(i1, i2) << endl;  // inline request to expand to (i1 > i2) ? i1 : i2
13   return 0;
14 }
```

The compiler may expand line 11 as:

```
cout << (i1 > i2) ? i1 : i2 << endl;
```

The expanded statement is faster than invoking a function call. The trade-off is bigger code size.

Inline is just a recommendation. Compiler may not support or may ignored the recommendation.

Inline functions and Macros

In C, you can use the #define preprocessor directive to define a macro with an argument, which would then be expanded during pre-processing.

For example,

```cpp
#include <iostream>
using namespace std;

#define SQUARE(x) x*x     // Macro with argument

inline int square(int x)
{
  return x*x;
}  // inline function

int main()
{
  cout << SQUARE(5) << endl;  // expand to 5*5 (25)
  int x = 2, y = 3;
  cout << SQUARE(x) << endl;  // expand to x*x (4)

  // Problem with the following macro expansions
  cout << SQUARE(5+5) << endl;   // expand to 5+5*5+5 -
wrong answer
  cout << square(5+5) << endl;   // Okay square(10)
  cout << SQUARE(x+y) << endl;   // expand to x+y*x+y -
wrong answer
  cout << square(x+y) << endl;   // Okay
  // can be fixed using #define SQUARE(x) (x)*(x)

  cout << SQUARE(++x) << endl;   // expand to ++x*++x
(16) - x increment twice
  cout << x << endl;             // x = 4
  cout << square(++y) << endl;   // Okay ++y, (y*y) (16)
  cout << y << endl;             // y = 4
}
```

Inline function is preferred over macro expansion, as seen from the above example.

Command-Line Arguments

You may include arguments in the command-line, when running a program, for example,

```
$ gcc -o test test.cpp
```

"-o test test.cpp" are called command-line arguments. Each of the arguments is a string, all the arguments form a string array, and passed into the main() function of the program.

To process command-line argument, the main() function shall use this header:

```
int main(int argc, char *argv[]) { ...... }
```

The second parameter char *argv[] captures the string array, while the first parameter captures the size of the array or the number of arguments.

For example

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char *argv[])
{
 if (argc != 4)
 {
   cout << "Usage: Arithmetic num1 num2 operator" << endl;
   exit(1);
 }

 int operand1 = atoi(argv[1]); // Parse string to int
 int operand2 = atoi(argv[2]); // Parse string to int
 char op = argv[3][0];         // Extract first character only

 switch (op) {
   case '+':
     cout << operand1 << " + " << operand2 << " = " << operand1 +
operand2 << endl;
     break;
   case '-':
     cout << operand1 << " - " << operand2 << " = " << operand1 -
operand2 << endl;
     break;
   case '*':
     cout << operand1 << " * " << operand2 << " = " << operand1 *
operand2 << endl;
     break;
   case '/':
     cout << operand1 << " / " << operand2 << " = " << operand1 /
operand2 << endl;
     break;
   default:
     cout << "Unknown operator" << endl;
     exit(1);
 }

 return 0;
}
```