Compiler Optimization Techniques

Contact Session-16

vivekanandamr@wilp.bits-pilani.ac.in

BITS Pilani
Pilani Campus

Code Optimization in Compiler Design

- A program transformation technique, which tries to improve the intermediate code by making it consume fewer resources, so that faster-running machine code will result.
- ***Resources ?????***
- ***Optimizing process should meet the following objectives :***
- *The optimization must be correct - **meaning of the program** must not be changed.*
- *should **increase** the **speed** and **performance** of the program*
- ***compilation time** must be kept **reasonable**.*
- *optimization process **should not delay** the overall compiling process.*

Code Optimization in Compiler Design

- **When to Optimize?**
- Optimization of the code is often performed at **the end of the development stage** since it reduces readability and adds code that is used to increase the performance.
- ***Why Optimize?***
- *Reduce the space consumed and increases the speed of compilation.*
- *An optimized code often promotes re-usability.*
- *manually performing the optimization is also tedious and is better done using a code optimizer.*

Compiler Optimization Levels

- **4 numeric levels**
- On where n ranges from 0 (no optimization) to 3.
- -O0 (do no optimization, the default if no optimization level is specified)
- -O1 (optimize minimally)
- -O2 (optimize more)
- -O3 (optimize even more)

Compiler Optimization Levels

- **4 numeric levels**
- On where n ranges from 0 (no optimization) to 3.
- **O0 Optimization**
- O0 - No optimization (the default), generates an optimized code but has the fastest compilation time.
- Many compilers do substantial optimization even if no optimization is specified. O0 means (almost) no optimization.
- Each source code command relates directly to the corresponding instructions in the executable file.

Compiler Optimization Levels

- **4 numeric levels**
- On where n ranges from 0 (no optimization) to 3.
- **O1 Optimization**
- Minimal impact on compilation time as compared to O0 compile.
- This enables most common forms of optimization that requires no size versus speed decisions, including function inlining.
- Only optimizations applied to straight line code (basic blocks) like instruction scheduling.

Compiler Optimization Levels

- **4 numeric levels**
- On where n ranges from 0 (no optimization) to 3.
- **O2 Optimization**
- Optimizations that always increase performance.
- Can significantly increase compilation time.
- O2 optimization examples:
  - Loop nest optimization.
  - Global optimization within a function scope.
  - 2 passes of instruction scheduling.
  - Dead code elimination.
  - Global register allocation.

Compiler Optimization Levels

- **4 numeric levels**
- On where n ranges from 0 (no optimization) to 3.
- **O3 Optimization**
- More extensive optimizations that may in some cases slow down performance.
- Optimizes loop nests rather than just inner loops, i.e. inverts indices, etc.
- "Safe" optimizations – produces answers identical with those produced by – O0.

Compiler Optimization Levels


```
#include <stdio.h>
double powern (double d, unsigned n)
{
  double x = 1.0;
  unsigned j;
  for (j = 1; j <= n; j++)
    x *= d;
  return x;
}
```

```
int main (void)
{
  double sum = 0.0;
  unsigned i;
   for (i = 1; i <= 100000000; i++)
    {
      sum += powern (i, i % 5);
    }
  printf ("sum = %g\n", sum);
  return 0;
}
```

```
$ gcc -Wall -O0 test.c -lm
$ time ./a.out
real    0m13.388s
user    0m13.370s
sys     0m0.010s
```

```
$ gcc -Wall -O1 test.c -lm
$ time ./a.out
real    0m10.030s
user    0m10.030s
sys     0m0.000s
```

```
$ gcc -Wall -O2 test.c -lm
$ time ./a.out
real    0m8.388s
user    0m8.380s
sys     0m0.000s
```

```
$ gcc -Wall -O3 test.c -lm
$ time ./a.out
real    0m6.742s
user    0m6.730s
sys     0m0.000s
```

User: CPU time spent running the process

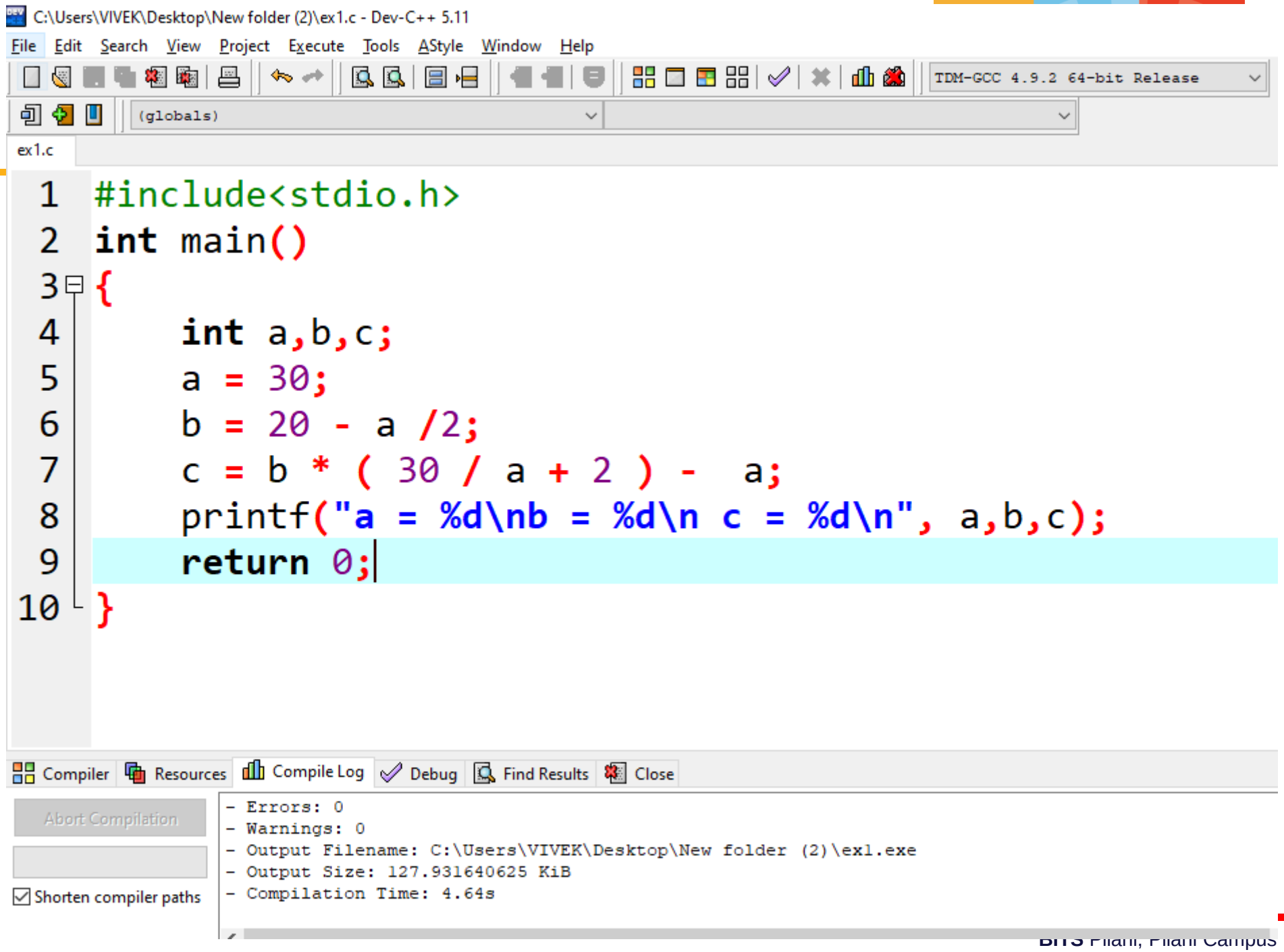
Compiler Optimizations

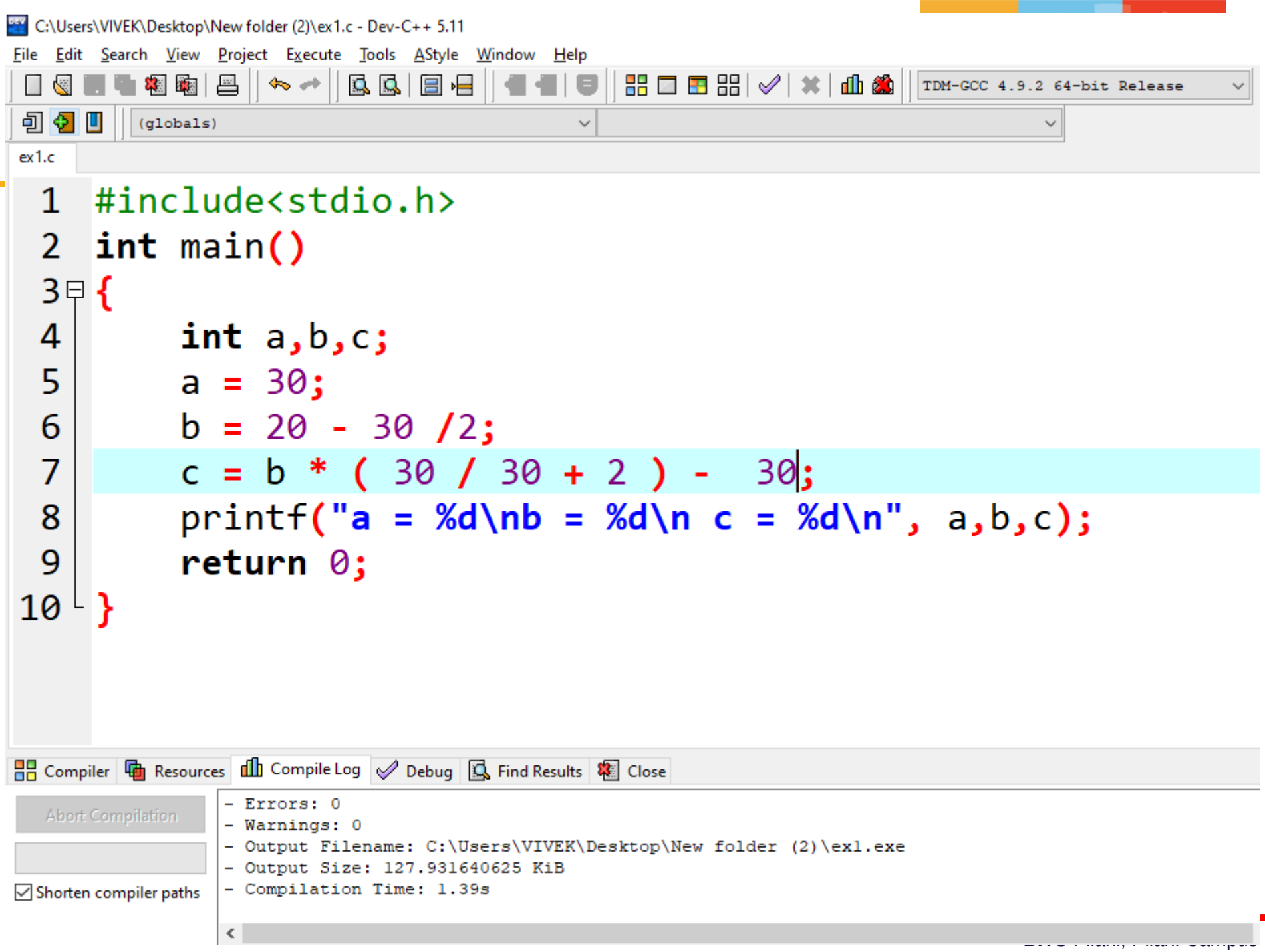
- Machine independent (apply equally well to most CPUs)
  - Constant propagation
  - Constant folding
  - Common Sub expression Elimination
  - Dead Code Elimination
  - Loop Invariant Code Motion
  - Function In lining
- Machine dependent (apply differently to different CPUs)
  - Instruction Scheduling
  - Loop unrolling
  - Parallel unrolling
- Could do these manually, better if compiler does them
  - Many optimizations make code less readable/maintainable

Compiler Optimizations

innovate achieve lead

- **Compile Time Evaluation - Constant propagation**
- One of the local code optimization technique
- Process of replacing the constant value of variables in the expression during compile time.
- Constants assigned to a variable can be propagated through the flow graph and can be replaced when the variable is used.
- **Example:**
- Suppose we are using pi variable and assign it value of 22/7
- pi = 22/7 = 3.14
- Compiler has to first perform **division** operation
- ***Assign*** the computed result 3.14 to the variable pi
- Length = (22/7)*d


C:\Users\VIVEK\Desktop\New folder (2)\ex1.c - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Release
(globals)
ex1.c
1 #include<stdio.h>
2 int main()
3 {
4 int a,b,c;
5 a = 30;
6 b = 20 - a /2;
7 c = b * (30 / a + 2) - a;
8 printf("a = %d\nb = %d\n c = %d\n", a,b,c);
9 return 0;
10 }
Compiler Resources Compile Log Debug Find Results Close
Abort Compilation
Shorten compiler paths
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\VIVEK\Desktop\New folder (2)\ex1.exe
- Output Size: 127.931640625 KiB
- Compilation Time: 4.64s


C:\Users\VIVEK\Desktop\New folder (2)\ex1.c - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Release
(globals)
ex1.c
1 #include<stdio.h>
2 int main()
3 {
4 int a,b,c;
5 a = 30;
6 b = 20 - 30 /2;
7 c = b * (30 / 30 + 2) - 30;
8 printf("a = %d\nb = %d\n c = %d\n", a,b,c);
9 return 0;
10 }
Compiler Resources Compile Log Debug Find Results Close
Abort Compilation
Shorten compiler paths
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\VIVEK\Desktop\New folder (2)\ex1.exe
- Output Size: 127.931640625 KiB
- Compilation Time: 1.39s

Compiler Optimizations

- **Constant propagation**
- **Consider the following pseudocode:**
  - int x = 14;
  - int y = 7 - x / 2;
  - return y * (28 / x + 2);
- Propagating x yields:
  - int x = 14;
  - int y = 7 - 14 / 2;
  - return y * (28 / 14 + 2);
- Continuing to propagate yields the following
  - int x = 14;
  - int y = 0;
  - return 0;

C:\Users\VIVEK\Desktop\New folder (2)\ex2.c - Dev-C++ 5.11

File Edit Search View Project Execute Tools AStyle Window Help

TDM-GCC 4.9.2 64-bit Release

(globals)

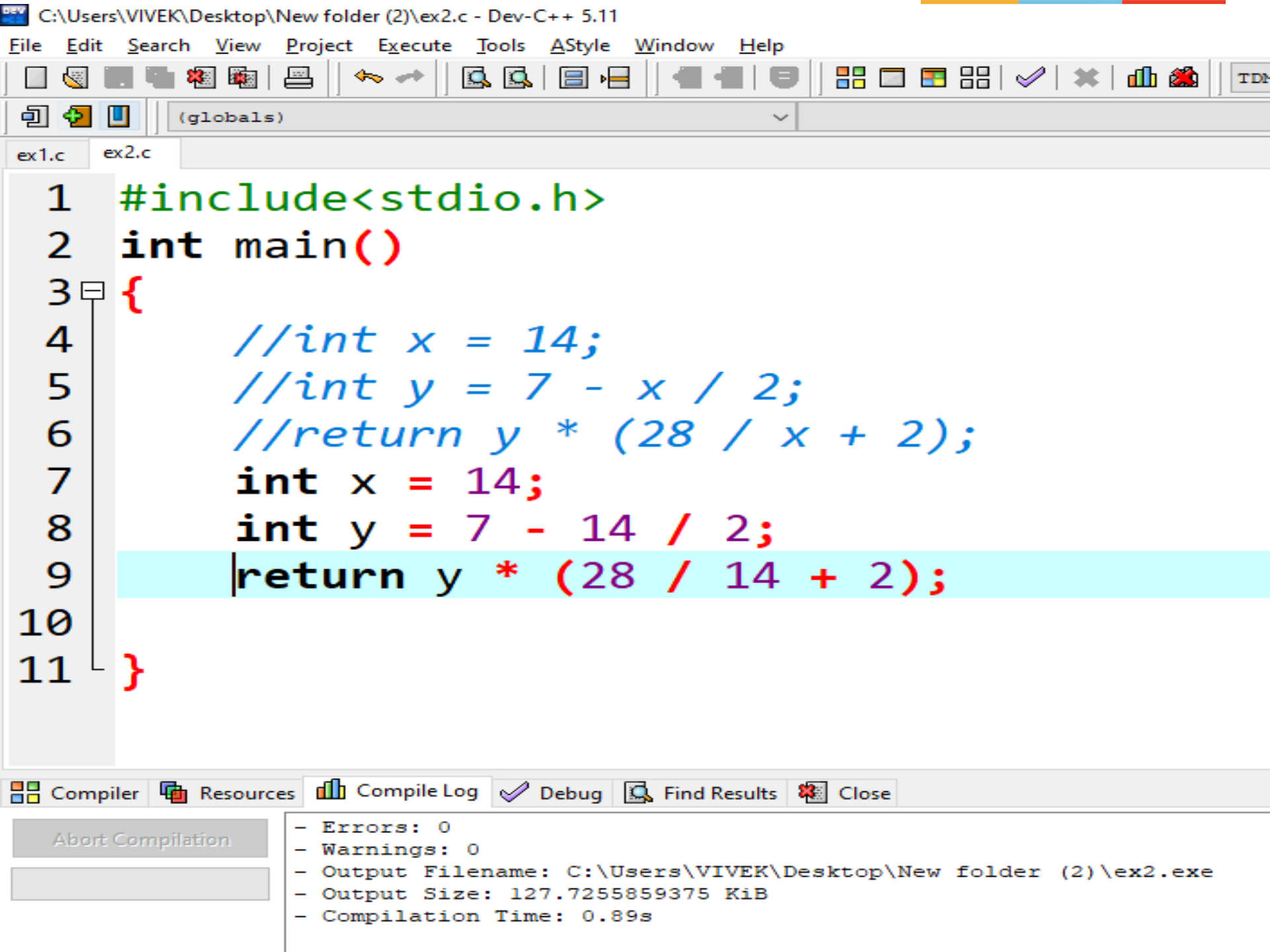
ex1.c ex2.c

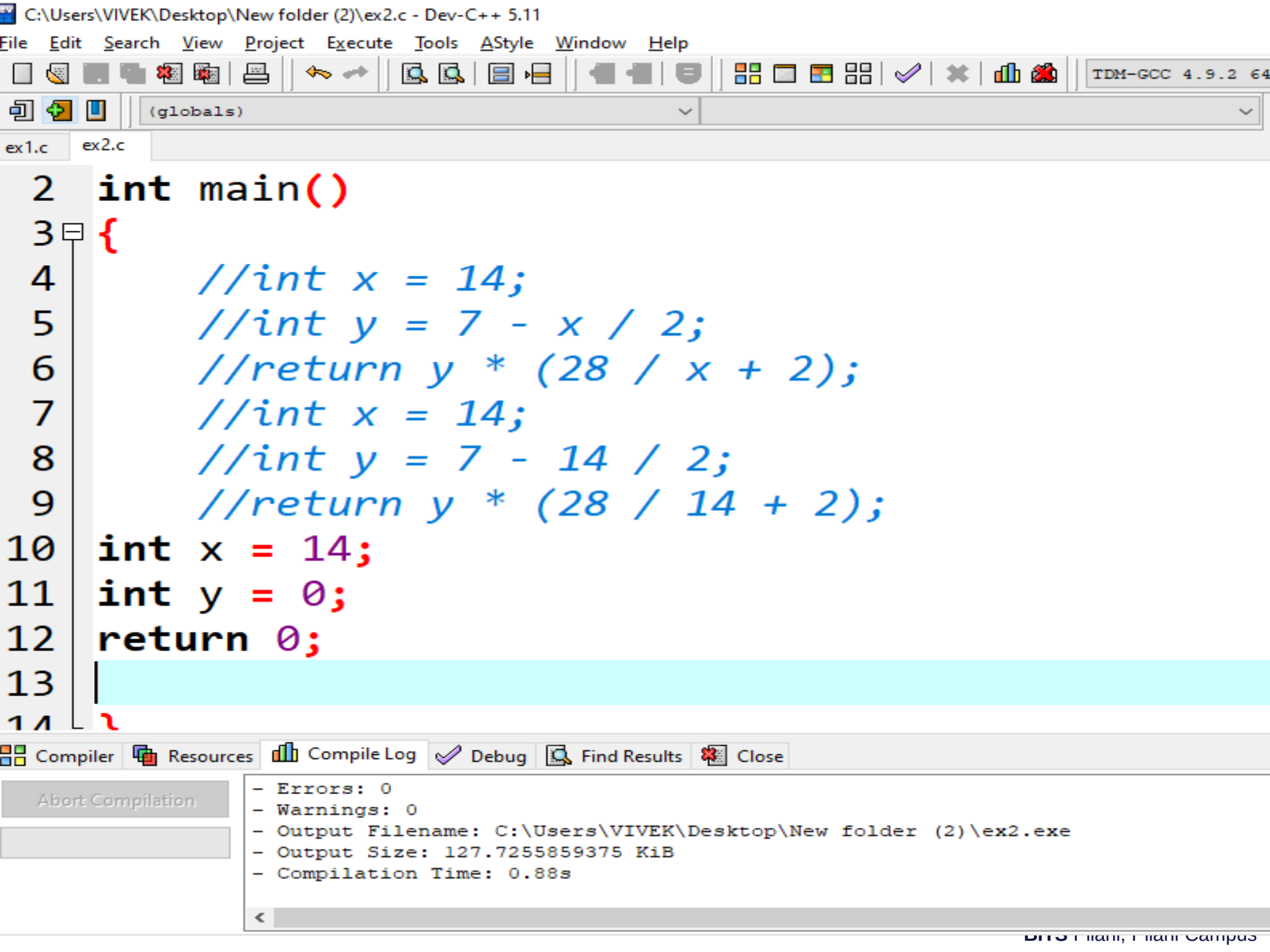
```
1  #include<stdio.h>
2  int main()
3  {
4      int x = 14;
5      int y = 7 - x / 2;
6      return y * (28 / x + 2);
7  }
```

Compiler Resources Compile Log Debug Find Results Close

Abort Compilation

```
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\VIVEK\Desktop\New folder (2)\ex2.exe
- Output Size: 127.7255859375 KiB
- Compilation Time: 1.30s
```


C:\Users\VIVEK\Desktop\New folder (2)\ex2.c - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
(globals)
ex1.c
ex2.c
1 #include<stdio.h>
2 int main()
3 {
4 //int x = 14;
5 //int y = 7 - x / 2;
6 //return y * (28 / x + 2);
7 int x = 14;
8 int y = 7 - 14 / 2;
9 return y * (28 / 14 + 2);
10
11 }
Compiler Resources Compile Log Debug Find Results Close
Abort Compilation
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\VIVEK\Desktop\New folder (2)\ex2.exe
- Output Size: 127.7255859375 KiB
- Compilation Time: 0.89s

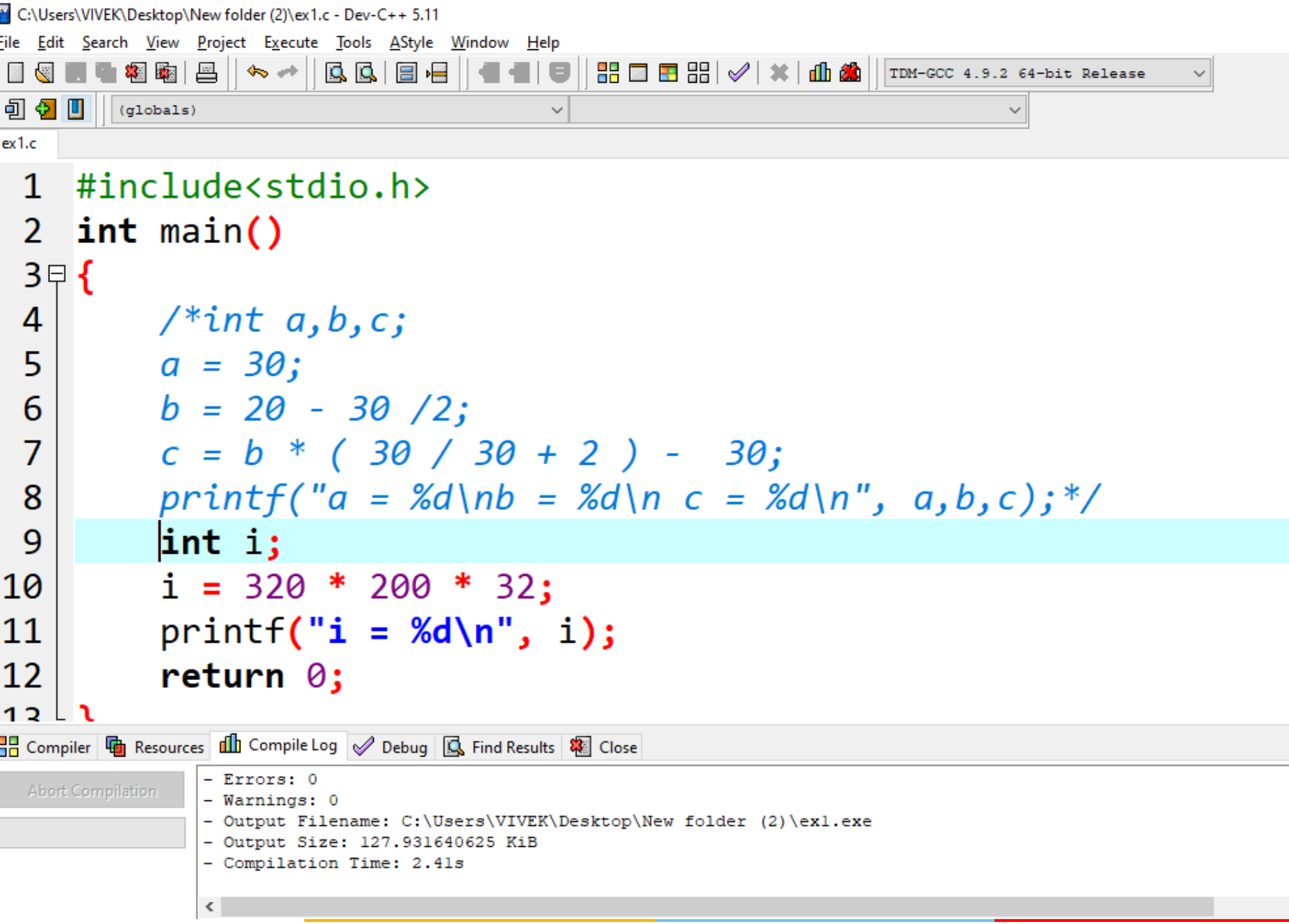

C:\Users\VIVEK\Desktop\New folder (2)\ex2.c - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64
(globals)
ex1.c
ex2.c
2 int main()
3 {
4 //int x = 14;
5 //int y = 7 - x / 2;
6 //return y * (28 / x + 2);
7 //int x = 14;
8 //int y = 7 - 14 / 2;
9 //return y * (28 / 14 + 2);
10 int x = 14;
11 int y = 0;
12 return 0;
13
Compiler Resources Compile Log Debug Find Results Close
Abort Compilation
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\VIVEK\Desktop\New folder (2)\ex2.exe
- Output Size: 127.7255859375 KiB
- Compilation Time: 0.88s

Compiler Optimizations

- **Compile Time Evaluation - Constant folding**
- Process of **recognizing and evaluating** constant expressions at **compile time** rather than computing them at runtime.
- **Terms in constant expressions are typically** simple literals(integer literal 2), but they may also be variables whose values are known at compile time.
- Consider the statement:
- i = 320 * 200 * 32;

Compiler Optimizations

- **Constant folding**
- Constant folding is an optimization technique that eliminates expressions that calculate a value that can already be determined before code execution.
- These are typically calculations that only reference constant values or expressions that reference variables whose values are constant.
- For example, consider the statement:
- i <- 320 * 200 * 32
- Most compilers would not actually generate two multiply instructions. Instead, they identify constructs such as these and substitute the computed values (in this case, 2048000). This way, the code will be replaced by:
- i <- 2048000


C:\Users\VIVEK\Desktop\New folder (2)\ex1.c - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Release
(globals)
ex1.c
1 #include<stdio.h>
2 int main()
3 {
4 /*int a,b,c;
5 a = 30;
6 b = 20 - 30 /2;
7 c = b * (30 / 30 + 2) - 30;
8 printf("a = %d\nb = %d\n c = %d\n", a,b,c);*/
9 int i;
10 i = 320 * 200 * 32;
11 printf("i = %d\n", i);
12 return 0;
Compiler Resources Compile Log Debug Find Results Close
Abort Compilation
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\VIVEK\Desktop\New folder (2)\ex1.exe
- Output Size: 127.931640625 KiB
- Compilation Time: 2.41s

C:\Users\VIVEK\Desktop\New folder (2)\ex1.c - Dev-C++ 5.11

File Edit Search View Project Execute Tools AStyle Window Help

TDM-GCC 4.9.2 64-bit Release

(globals)

ex1.c

```
2  int main()
3  {
4      /*int a,b,c;
5      a = 30;
6      b = 20 - 30 /2;
7      c = b * ( 30 / 30 + 2 ) -  30;
8      printf("a = %d\nb = %d\n c = %d\n", a,b,c);*/
9      int i;
10     i = 2048000;
11     printf("i = %d\n", i);
12     return 0;
13 }
```

Compiler Resources Compile Log Debug Find Results Close

Abort Compilation

```
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\VIVEK\Desktop\New folder (2)\ex1.exe
- Output Size: 127.931640625 KiB
- Compilation Time: 1.02s
```

Compiler Optimizations

- **Constant folding**
- To-Do
- **Implement intelligent constant folding?**
- For example: fold 0 * x to 0
- However, this could change code semantics, in the second case, if x does not exist then the code would not throw an error, meanwhile, in the first case, it would.
- **Reorder variables with associative operators?**
- The R parser has left associativity, so x + 10 + 200 is ((x + 10) + 200). So this is not being folded to x + 210.
- If we consider operators with associativity, we could replace x + 10 + 200 to 10 + 200 + x, and then fold it to 210 + x

Compiler Optimizations

- Constant folding and propagation are typically used together to achieve many simplifications and reductions, by interleaving them iteratively until no more changes occur.
- Consider this unoptimized pseudocode that returns a random number

```
int a = 30;
int b = 9 - (a / 5);
int c;

c = b * 4;
if (c > 10) {
   c = c - 10;
}
return c * (60 / a);
```

```
int a = 30;
int b = 3;
int c;

c = b * 4;
if (c > 10) {
   c = c - 10;
}
return c * 2;
```

```
int a = 30;
int b = 3;
int c;

c = 12;
if (true) {
   c = 2;
}
return c * 2;
```

```
int c;
c = 12;

if (true) {
   c = 2;
}
return c * 2;
```

```
return 4;
```

Compiler Optimizations

- **Dead Code Elimination (DCE)** [dead code removal, dead code stripping, or dead code strip]
- A compiler optimization to remove code which does not affect the program results.
- Code that is unreachable or that does not affect the program (e.g. dead stores) can be eliminated.
- **DCE benefits:**
- Shrinks program size- important consideration in some contexts
- It allows the running program to avoid executing irrelevant operations - reduces its running time.
- It enable further optimizations by simplifying program structure.

Compiler Optimizations

- **Dead Code Elimination (DCE)**
- Dead code includes:
- Code that can never be executed (unreachable code), and
- Code that only affects dead variables (written to, but never read again), that is, irrelevant to the program.

Compiler Optimizations

innovate achieve lead

```
int global;
void f ()
{
  int i;
  i = 1;     /* dead store */
  global = 1; /* dead store */
  global = 2;
  return;
  global = 3; /* unreachable */
}
```

the code fragment after dead code elimination.

```
int global;
void f ()
{
  global = 2;
  return;
}
int foo(void)
{
  int a = 24;
  int b = 25; /* Assignment to dead variable
*/
  int c;
  c = a * 4;
  return c;
  b = 24; /* Unreachable code */
  return 0;
}
```

Compiler Optimizations


```
int foo(void)
{
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a * 4;
  return c;
  b = 24; /* Unreachable code */
  return 0;
}
```

- The value of b after the first assignment is not used inside foo
- b is declared as a local variable inside **foo**, so its value cannot be used outside **foo**
- Thus, the variable b is dead and an optimizer can reclaim its storage space and eliminate its initialization
- Because the first return statement is executed unconditionally, no feasible execution path reaches the second assignment to b

```
int main(void) {
   int a = 5;
   int b = 6;
   int c;
   c = a * (b / 2);
   if (0) {   /* DEBUG */
     printf("%d\n", c);
   }
   return c;
 }
```

- The expression 0 will always evaluate to false, the code inside the if statement can never be executed, and dead code elimination would remove it entirely from the optimized program.
- Using an optimizer with dead code elimination eliminates the need for using a preprocessor to perform the same task.

Compiler Optimizations

- **Common Sub-expression Elimination (CSE)**
- An expression is a Common Subexpression (CSE) if the expression was previously computed and the values of the operands have not changed since the previous computation.
- Recomputing the expression can be eliminated by using the value of the previous computation.

```
a = c * d;
    :
    :
d = (c * d + t) * u
```

→

```
a = c * d;
    :
d = (a + t) * u
```

- Try to only compute a given expression once (assuming the variables have not been modified)

Compiler Optimizations

- **Common Sub-expression Elimination (CSE)**
  - In the code fragment below, the second computation of the expression (x + y) can be eliminated.

```
i = x + y + 1;
j = x + y;
```

  - After CSE Elimination, the code fragment is rewritten as follows.

```
t1 = x + y;
i = t1 + 1;
j = t1;
```

Compiler Optimizations

- **Loop Invariant Code Motion (LICM)**
- Consists of statements or expressions which can be moved outside the body of a loop without affecting the semantics of the program

```
for (i=0; i < 100 ;  ++i) {
        a =5;
  for (j=0; j < 100 ; ++j) {
        b = 6;
    for (k=0 ; k < 100 ; ++k) {
      x= i+j+k;
    }
  }
}
```

⇒

```
a = 5;
b = 6;
for (i = 0; i < 100 ; ++i) {
  for (j = 0; j < 100 ; ++j) {
    for (k = 0 ; k < 100 ; ++k) {
        x= i+j+k;
    }
  }
}
```

- Loop invariant: value does not change across iterations
- LICM: move invariant code out of the loop
- Big performance wins:
  - Inner loop will execute 1,000,000 times
  - Moving code out of inner loop results in big savings

Compiler Optimizations

- **Loop Invariant Code Motion (LICM)**
- Consists of statements or expressions which can be moved outside the body of a loop without affecting the semantics of the program

```
for (i=0; i < 100 ;  ++i) {
        X= Y+Z+K;
        a[i] = 6*i;

}
```

⇒

```
X= Y+Z+K;
for (i=0; i < 100 ;  ++i) {
        a[i] = 6*i;

}
```

- Analysis
- • a computation is done within a loop and
- • result of the computation is the same as long as we keep going around the loop
- • Transformation
- • move the computation outside the loop

Compiler Optimizations

- **Loop Invariant Code Motion (LICM)**
- In the following code sample, two optimizations can be applied.

```
int i = 0;
while (i < n) {
    x = y + z;
    a[i] = 6 * i + x * x;
    ++i;
}
```

```
int i = 0;
if (i < n) {
    x = y + z;
    int const t1 = x * x;
    do {
        a[i] = 6 * i + t1;
        ++i;
    } while (i < n);
}
```

Compiler Optimizations

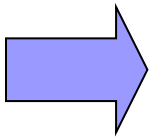
innovate achieve lead

- **Function Inlining**
- The overhead associated with calling and returning from a function can be eliminated by expanding the body of the function inline, and additional opportunities for optimization may be exposed as well.
- Most of the Inline functions are used for small computations. They are not suitable for large computing.

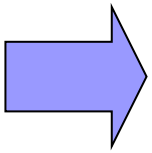
```
#include<stdio.h>
inline int mul(int a, int b) //inline function declaration{
   return(a*b);
}
int main(){
   int c;
   c=mul(2,3);
   printf("Multiplication:%d\n",c);
   return 0;
}
```

Function In-lining


```
foo(int z){
  int m = 5;
  return z + m;
}main(){
   ...
   x = foo(x);
   ...
 }
```



```
main(){
 ...
 {
  int foo_z = x;
  int m = 5;
  int foo_return = foo_z + m;
  x = foo_return;
 }
 ...
}
```



```
main(){
  ...
  x = x + 5;
  ...
}
```

Code size

- can decrease if small procedure body and few calls
- can increase if big procedure body and many calls

Performance

- eliminates call/return overhead
- can expose potential optimizations
- can be hard on instruction-cache if many copies made

As a Programmer:

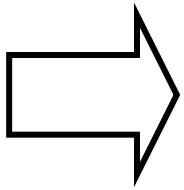
- a good compiler should inline for best performance
- feel free to use procedure calls to make your code readable!

Loop Unrolling

- Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program.
- It basically remove or reduce iterations.
- Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

```
#include<stdio.h>
int main()
{
        int i;
        for(i=0;i<5;i++)
        printf("Hello");
        return 0;

}
```

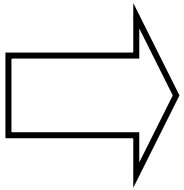


```
#include<stdio.h>
int main()
{
        printf("Hello");
        printf("Hello");
        printf("Hello");
        printf("Hello");
        printf("Hello");
        return 0;

}
```

Loop Unrolling

```
j = 0;
while (j < 100){
    a[j] = b[j+1];
    j += 1;
}
```



```
j = 0;
while (j < 99){
    a[j] = b[j+1];
    a[j+1] = b[j+2];
    j += 2;
}
```

reduces loop overhead

- Fewer adds to update j
- Fewer loop condition tests

enables more aggressive instruction scheduling

- more instructions for scheduler to move around

Loop Unrolling

Advantages:

- Increases program efficiency.
- Reduces loop overhead.
- If statements in loop are not dependent on each other, they can be executed in parallel.

Disadvantages:

- Increased program code size, which can be undesirable.
- Possible increased usage of register in a single iteration to store temporary variables which may reduce performance.
- Apart from very small and simple codes, unrolled loops that contain branches are even slower than recursions.

Example Compiler: gcc Optimization Levels

-g:
- Include debug information, no optimization

-O0:
- Default, no optimization

-O1:
- Do optimizations that don't take too long
- CP, CF, CSE, DCE, LICM, inlining small functions

-O2:
- Take longer optimizing, more aggressive scheduling

-O3:
- Make space/speed trade-offs: loop unrolling, more inlining

-Os:
- Optimize program size

Performance Optimization: Requirements

1) Preserve correctness
   - the speed of an incorrect program is irrelevant
2) On average improve performance
   - Optimized may be worse than original if unlucky
3) Be "worth the effort"
   - Is this example worth it?
     - 1 person-year of work to implement compiler optimization
     - 2x increase in compilation time
     - 0.1% improvement in speed

How do optimizations improve performance?

Execution_time = num_instructions * CPI

- Fewer cycles per instruction
  - Schedule instructions to avoid dependencies
  - Improve cache/memory behavior
    - Eg., locality
- Fewer instructions
  - Eg: Target special/new instructions

Role of Optimizing Compilers

Provide efficient mapping of program to machine

- eliminating minor inefficiencies
- code selection and ordering
- register allocation

Don't (usually) improve asymptotic efficiency

- up to programmer to select best overall algorithm
- big-O savings are (often) more important than constant factors
  - but constant factors also matter

Limitations of Optimizing Compilers

- Operate Under Fundamental Constraints
  - Must not cause any change in program behavior under any possible condition
- Most analysis is performed only within procedures
  - inter-procedural analysis is too expensive in most cases
- Most analysis is based only on *static* information
  - compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

Role of the Programmer

How should a programmer write programs, given that there is a good, optimizing compiler?

Don't: Smash Code into Oblivion

- Hard to read, maintain, & assure correctness

Do:

- Select best algorithm
- Write code that's readable & maintainable
  - Procedures, recursion
  - Even though these factors can slow down code
- Eliminate optimization blockers
  - Allows compiler to do its job

Focus on Inner Loops

- Do detailed optimizations where code will be executed repeatedly
- Will get most performance gain here

Thank you.

BITS Pilani

Pilani Campus