Trabalho Prático 2

Victor Augusto Hon Fonseca- número de matrícula: 2022035814

1.0 Introdução

Sob o contexto passado pelo enunciado do trabalho que estou trabalhando em uma indústria têxtil e meu chefe está tendo problemas para determinar os tamanhos das peças de tecido que devem ser compradas, e que eu devo determinar uma peça de tecido que consiga cobrir todos os pontos presentes na lista com um conjunto de pontos no plano cartesiano, eu devo implementar um sistema que consiga determinar o fecho convexo dos pontos passados. Para resolver esse problema, o programa receberá coordenadas de pontos de um arquivo “txt” e, a partir disso, usando estruturas de dados, classes e funções determinadas realizará essa funcionalidade requisitadas.

2.0 Método

O programa foi desenvolvido utilizando a linguagem de programação C++, as bibliotecas “iostream”, “string”, “cmath”(biblioteca equivalente à “math.h” da linguagem C), limits, stdexcept dessa linguagem para os arquivos, de modo geral, e, no arquivo “main.cpp”, foram utilizadas as bibliotecas “fstream”, para a leitura das linhas do arquivo “txt” utilizado e interpretação dos termos de cada linha como “_x” e “_y”, a biblioteca “iomanip”, para determinar quantas casas de precisão para alguns valores e “chrono” para determinar o tempo de execução de determinadas funções. Além disso, vale relatar que o programa foi compilado utilizando o compilador G++, da GNU Compiler Collection, na IDE Visual Studio Code, utilizando o recurso Subsistema do Windows para Linux (WSL), do sistema operacional Windows, na IDE Visual Studio Code.

2.1 Estruturas de Dados

É importante destacar que, durante a implementação do programa, além de classes, também foi necessário utilizar estruturas de dados auxiliares, para armazenar, por exemplo, objetos do tipo ponto. No meu programa, após as ordenadas ser lida, ela são usadas para criar um objeto do tipo ponto, o qual é armazenado dentro de uma estrutura de dados implementada por mim, chamada de “vector”, criada utilizando de templates, o que permite escolher o tipo de elemento armazenado dentro dela. Ademais, em seu interior há a classe iterador, o qual retorna um ponteiro para determinada posição do “vector”, sendo essa composta de atributos como capacidade, uma constante, índice para o último elemento e um array, outra estrutura de dados, mais simples, usada para armazenar elementos passados, e a classe “iterador” composta por um ponteiro como um atributo.

Ambas possuem métodos já esperados para essas estruturas de dados, entre eles: construtor e destrutor. Vale destacar que a classe “vector” possui métodos para adicionar elementos, retirar elementos, retornar o tamanho do vector, construir um vector a partir de iteradores e um iterador que aponta para sua primeira posição e outro para depois da última posição. Já a classe “iterator” possui métodos para permitir que operações de: adição aconteçam com os iteradores, fazendo com que apontem para posições posteriores, substração aconteçam com os iteradores, fazendo com que

apontem para posições anteriores e comparação aconteçam com os iteradores (retornando falso ou verdadeiro).

2.2 Classes

O programa utiliza apenas duas classes, uma é a classe Ponto2d, a qual possui como atributos: as ordenadas “x” e “y” e como métodos principais: funções para construir um objeto do tipo ponto caso sejam passadas ordenadas específicas ou não, para calcular a distância entre dois pontos, para calcular o ângulo entre dois pontos considerando o eixo X positivo e para calcular o ângulo entre dois pontos considerando o eixo X negativo e para imprimir os atributos do ponto. Já a outra classe é a “FechoConvexo, a qual possui como atributos: um vector de Pontos 2d chamado “_pontos”, que inicialmente armazena os pontos, mas depois o ponto de menor Y é retirado, um vector de Pontos 2d com o ponto de menor Y incluído, um vector de Pontos 2d chamado S e um Ponto 2d chamado p0, que é o com menor atributo Y.

2.3 Funções

O programa conta com 4 funções principais, todas sendo métodos da classe “FechoConvexo”, as quais são as responsáveis, em conjunto com funções auxiliares, para atender diretamente aos requisitos do trabalho:

-**GrahamInsertion**: Função que junto com o método de ordenação Insertion Sort, implementado na função “InsertionSort” (organiza os elementos da esquerda para a direita, um a um), a qual ordena os elementos de um vector de Pontos 2d, com base nos ângulos de cada um em relação ao ponto de menor Y(p0), por meio do algoritmo de Graham-Scan gera os pontos que formam o fecho convexo da lista de pontos passada.

-**GrahamMerge**: Função que junto com o método de ordenação Merge Sort (divide o vetor original em subvetores cada vez menores, ordena os subvetores dois a dois e , aos poucos, junta-os até novamente termos um vetor do tamanho do original), implementado por meio da associação das funções “MergeSort”, “MergeSort2” e “merge”, a qual ordena os elementos de um vector de Pontos 2d, com base nos ângulos de cada um em relação ao ponto de menor Y(p0), por meio do algoritmo de Graham-Scan gera os pontos que formam o fecho convexo da lista de pontos passada.

-**GrahamRadix**: Função que junto com o método de ordenação Radix Exchange Sort (transforma os elementos do vetor da base decimal para a base binária, no caso a comparação é feita entre os ângulos de cada ponto em relação ao p0, começa-se com um índice no final do vetor e outro no início, assume-se que no início todos os bits são 0 e no final 1, a cada par compara-se qual é maior e qual é menor, troca-se eles de posição, quando se cruzam, a função é chamada do início até o cruzamento e do cruzamento ao final e avança-se para o próximo bit, até se chegar ao último), implementado por meio da associação das funções “getBinaryDigit”, “radixsort” e “quicksortB”, a qual ordena os elementos de um vector de Pontos 2d, com base nos ângulos de cada um em relação ao ponto de menor Y(p0), por meio do algoritmo de Graham-Scan gera os pontos que formam o fecho convexo da lista de pontos passada.

-**Jarvis**: Função que junto com as funções JarvisRight(calcula a cadeia direita do fecho convexo, começando do ponto mais baixo e, em caso de empate, mais à direita, até o ponto mais alto e, em caso de empate, mais à direita, avançando escolhendo o ponto

com menor ângulo primeiro em relação ao ponto mais baixo, depois em relação a esse mesmo, e, assim por diante) e JarvisLeft(calcula a cadeia esquerda do fecho convexo, começando do ponto mais alto e, em caso de empate, mais à esquerda, até o ponto inicial de “JarvisRight”, avançando escolhendo o ponto com menor ângulo, em relação ao eixo negativo, primeiro em relação ao ponto mais alto mencionado, depois em relação a esse mesmo, e, assim por diante) calcula o fecho convexo de uma lista de pontos usando o algoritmo da “Marcha de Jarvis”.

3.0 Análise de Complexidade

3.1 Complexidade quanto ao tempo

-**GrahamInsertion**: Possui complexidade em relação ao tempo O(n^2), pois chama dentro dela a função “InsertionSort” que tem complexidade O(n^2) e dentro dela mesma só há um loop.

-**GrahamMerge**: Possui complexidade em relação ao tempo O(nlogn), pois chama dentro dela a função “merge” que tem complexidade O(n) e dentro dela as chamadas recursivas junto com o loop tornal a complexidade O(nlogn), já que dentro da própria função a complexidade é somente O(n).

-**GrahamRadix**: Possui complexidade em relação ao tempo O(n), pois chama dentro dela a função “quicksortB” que tem complexidade O(n), a função “radixSort” que tem complexidade O(n) e dentro da própria função a complexidade é somente O(n).

-**Jarvis**: Possui complexidade em relação ao tempo O(n^2), pois chama dentro dela a função “JarvisRight” que tem complexidade O(n^2), devido aos loops aninhados, a função “JarvisLeft” que tem complexidade O(n^2), devido aos loops aninhados e dentro da própria função a complexidade é somente O(n), devido aos loops que possui.

3.2 Complexidade quanto ao espaço

Considerando como forma de medida para a complexidade do espaço, o número de alocações dinâmicas realizadas, no meu programa, não há alocações dinâmicas realizadas, pois a única parte que utiliza de ponteiros corresponde à classe iterators, os quais não possuem um overhead de armazenamento significativo o suficiente, pois são objetos pequenos que contém geralmente um ponteiro. Além disso, essa classe está dentro da classe vector, logo, quando um vector é destruído, os objetos iterators também são. Logo, a complexidade espacial é O(1).

4.0 Estratégias de Robustez

Como o programa realiza a leitura dos pontos para os quais será calculado o fecho convexo de um arquivo “txt”, uma das principais estratégias de robustez aplicadas foi que caso o arquivo não possa ser aberto, seja porque ele não existe no diretório atual, ou por qualquer outra razão, lança-se uma exceção que interrompe a execução do programa imediatamente para evitar resultados inesperados, indicando que houve um erro na abertura do arquivo.

Além disso, dentro da classe “vector” (estrutura de dados criada), há várias exceções implementadas, como antes de inserir um elemento em um vector conferir se ele não está cheio, antes de remover um elemento de um vector conferir se ele não está vazio, ao tentar acessar um elemento do vector usando os operadores “[]”, conferir se a

posição existe mesmo e, por fim, na função erase conferir se o iterador passado não é inválido.

Ademais, a ferramenta de depuração valgriind foi usada para conferir se não havia vazamentos de memória no programa:

```
==29025==
==29025== HEAP SUMMARY:
==29025==     in use at exit: 0 bytes in 0 blocks
==29025==   total heap usage: 4 allocs, 4 frees, 82,392 bytes allocated
==29025==
==29025== All heap blocks were freed -- no leaks are possible
==29025==
==29025== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Assim, como a ferramenta gdb:

```
[Inferior 1 (process 30096) exited normally]
(gdb)
```

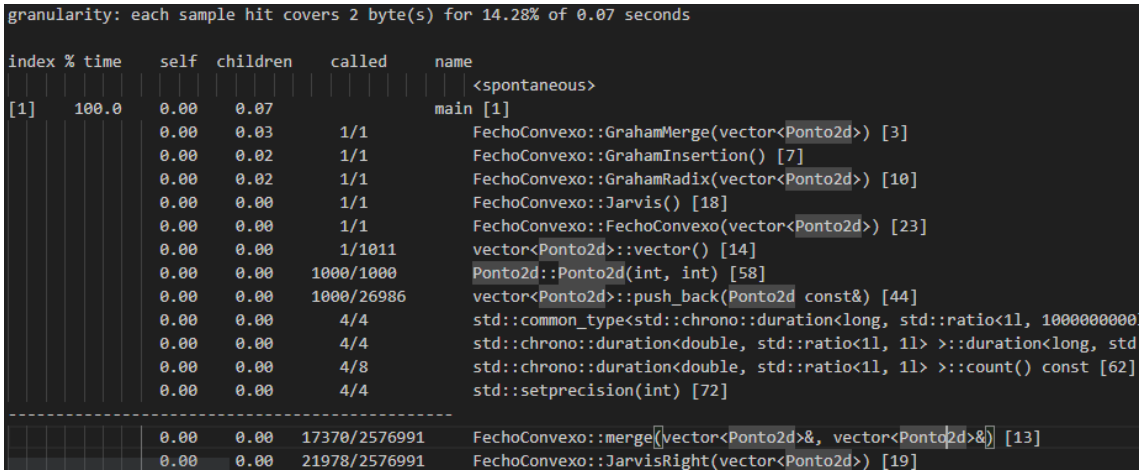
O que como pode-se perceber, indicou que não havia nenhum problema de gerenciamento de memória que poderia prejudicar o programa, contribuindo para sua robustez.

5.0 Análise Experimental

Para a minha análise experimental, uma das ferramentas utilizadas, foi o “gprof”, e o relatório “flat profile”, gerado a partir dessa ferramenta.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 42.89      0.03     0.03  9027029     0.00     0.00  Ponto2d::Ponto2d()
 42.89      0.06     0.03  2576991     0.00     0.00  Ponto2d::calcularAngulo(Ponto2d)
 14.30      0.07     0.01  2621050     0.00     0.00  Ponto2d::get_Y() const
  0.00      0.07     0.00  3827281     0.00     0.00  vector<Ponto2d>::operator[](int)
  0.00      0.07     0.00  3650389     0.00     0.00  Ponto2d::get_X() const
  0.00      0.07     0.00  1549423     0.00     0.00  vector<Ponto2d>::size() const
  0.00      0.07     0.00   110383     0.00     0.00  vector<Ponto2d>::iterator::operator<=(vector<Ponto2d>::
  0.00      0.07     0.00    64873     0.00     0.00  vector<Ponto2d>::iterator::operator*() const
  0.00      0.07     0.00    39569     0.00     0.00  FechoConvexo::getBinaryDigit(float, int)
  0.00      0.07     0.00    28226     0.00     0.00  vector<Ponto2d>::iterator::iterator(Ponto2d*)
  0.00      0.07     0.00    27968     0.00     0.00  vector<Ponto2d>::iterator::operator==(vector<Ponto2d>::
  0.00      0.07     0.00    26986     0.00     0.00  vector<Ponto2d>::push_back(Ponto2d const&)
  0.00      0.07     0.00    23200     0.00     0.00  vector<Ponto2d>::iterator::operator!=(vector<Ponto2d>::
  0.00      0.07     0.00    20453     0.00     0.00  vector<Ponto2d>::iterator::operator++()
  0.00      0.07     0.00    18101     0.00     0.00  vector<Ponto2d>::iterator::operator++(int)
```

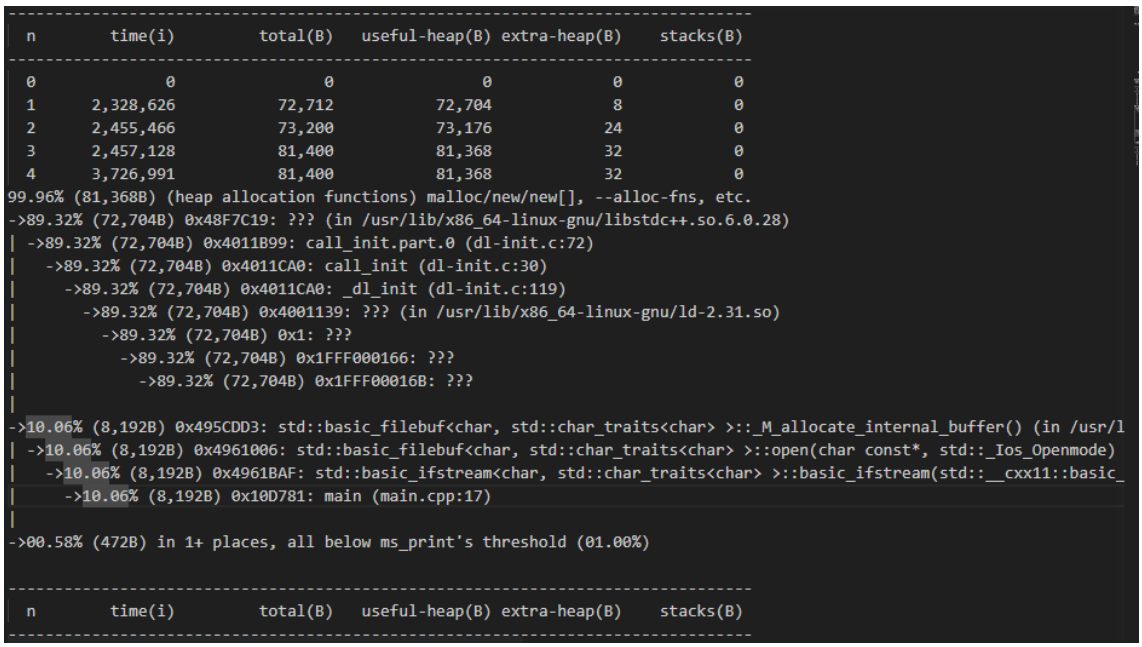


```
granularity: each sample hit covers 2 byte(s) for 14.28% of 0.07 seconds

index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00    0.07                 main [1]
                0.00    0.03       1/1           FechoConvexo::GrahamMerge(vector<Ponto2d>) [3]
                0.00    0.02       1/1           FechoConvexo::GrahamInsertion() [7]
                0.00    0.02       1/1           FechoConvexo::GrahamRadix(vector<Ponto2d>) [10]
                0.00    0.00       1/1           FechoConvexo::Jarvis() [18]
                0.00    0.00       1/1           FechoConvexo::FechoConvexo(vector<Ponto2d>) [23]
                0.00    0.00       1/1011        vector<Ponto2d>::vector() [14]
                0.00    0.00    1000/1000        Ponto2d::Ponto2d(int, int) [58]
                0.00    0.00    1000/26986       vector<Ponto2d>::push_back(Ponto2d const&) [44]
                0.00    0.00       4/4           std::common_type<std::chrono::duration<long, std::ratio<1l, 1000000000
                0.00    0.00       4/4           std::chrono::duration<double, std::ratio<1l, 1l> >::duration<long, std
                0.00    0.00       4/8           std::chrono::duration<double, std::ratio<1l, 1l> >::count() const [62]
                0.00    0.00       4/4           std::setprecision(int) [72]
-----------------------------------------------
                0.00    0.00   17370/2576991     FechoConvexo::merge(vector<Ponto2d>&, vector<Ponto2d>&) [13]
                0.00    0.00   21978/2576991     FechoConvexo::JarvisRight(vector<Ponto2d>) [19]
```

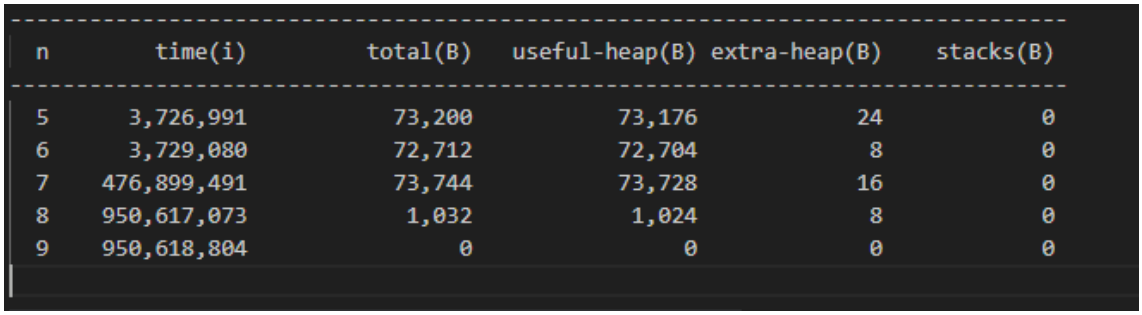
Essas imagens mostram apenas trechos do relatório “flat profile” gerado pelo gprof (o relatório era muito grande e todos os tempos variavam eram ou 0.01 ou 0.07, mas o relatório completo vai estar no diretório raiz do programa). Esse resultado curiosamente mostra todos os tempos como entre 0.01 e 0.07, como a ferramenta foi utilizada corretamente, compilando-se o código utilizando a flag “-pg” e a geração do relatório também, uma das possíveis causas que explica esse curioso comportamento, que contrasta com o elevado número de chamadas de cada operação, presente na primeira foto é que as operações realizadas são mais simples, não demandando um tempo de Cpu gasto significativo. Isso pode ser explicado pelo fato de que não há alocações dinâmicas significativas realizadas, as estruturas de dados usadas não possuem um grande espaço de armazenamento (3000) e que embora haja loops em algumas funções, a complexidade delas não ultrapassa O(n^2).

Como já foi previamente dito, não há problemas de gerenciamento de memória para afetar o desempenho do programa, o que constribui para sua otimização.



```
--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0              0            0            0
  1      2,328,626           72,712         72,704            8            0
  2      2,455,466           73,200         73,176           24            0
  3      2,457,128           81,400         81,368           32            0
  4      3,726,991           81,400         81,368           32            0
99.96% (81,368B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->89.32% (72,704B) 0x48F7C19: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.28)
| ->89.32% (72,704B) 0x4011B99: call_init.part.0 (dl-init.c:72)
|   ->89.32% (72,704B) 0x4011CA0: call_init (dl-init.c:30)
|     ->89.32% (72,704B) 0x4011CA0: _dl_init (dl-init.c:119)
|       ->89.32% (72,704B) 0x4001139: ??? (in /usr/lib/x86_64-linux-gnu/ld-2.31.so)
|         ->89.32% (72,704B) 0x1: ???
|           ->89.32% (72,704B) 0x1FFF000166: ???
|             ->89.32% (72,704B) 0x1FFF00016B: ???
|
->10.06% (8,192B) 0x495CDD3: std::basic_filebuf<char, std::char_traits<char> >::_M_allocate_internal_buffer() (in /usr/l
| ->10.06% (8,192B) 0x4961006: std::basic_filebuf<char, std::char_traits<char> >::open(char const*, std::_Ios_Openmode)
|   ->10.06% (8,192B) 0x4961BAF: std::basic_ifstream<char, std::char_traits<char> >::basic_ifstream(std::__cxx11::basic_
|     ->10.06% (8,192B) 0x10D781: main (main.cpp:17)
|
->00.58% (472B) in 1+ places, all below ms_print's threshold (01.00%)

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
```



n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
5	3,726,991	73,200	73,176	24	0
6	3,729,080	72,712	72,704	8	0
7	476,899,491	73,744	73,728	16	0
8	950,617,073	1,032	1,024	8	0
9	950,618,804	0	0	0	0

Ademais, utilizou-se ferramentas de profiling, como o Valgrind Massif, para visualizar o uso da memória ao longo do tempo, como se vê acima, permitindo, por exemplo a visualização de número de acessos e acesso a endereços. Como pode-se perceber alguns dos destaques de número de acessos foram no “main.cpp” ligado à biblioteca “fstream” e chamadas de alocação de heap como “malloc”, “new” e “new[]” no heap e, grande parte do que acontece dentro do heap em uma função não identificada dentro de “libstdc++.so.6.0.28”, o que indica que essa alocação de memória está relacionada ao funcionamento interno da biblioteca padrão do c++.

6.0 Conclusões

Nesse trabalho, foi requisitado que fosse implementado um sistema que indicasse o fecho convexo de listas de pontos 2d. Essa proposta foi um excelente exercício para o treinamento na parte de manipulação de dados bidimensionais e geometria computacional. Além disso, foi muito útil para a revisão de conteúdos mais antigos do curso como alocação dinâmica de memória e elaboração de classes, possibilitando com que estruturas de dados mais avançadas como “vector” utilizado fossem implementadas manualmente.

Entretanto, é importante frisar que um dos aprendizados mais importantes deste trabalho, foi sobre a reafirmação da importância do raciocínio lógico e abstrato para a solução de problemas, tendo sido necessário pensar em soluções criativas para realizar certas operações matemáticas. Um dos exemplos mais perceptíveis foi quando estava buscando implementar a classe vector, quando tive que implementar uma estrutura de dados até então bem mais avançada do que as que estava acostumado (pilha, fila circular, etc), devido, principalmente aos iteradores e na implementação da função para calcular ângulos com relação ao eixo X negativo.

Bibliografia

(Thomas H. *Cormen* - Algoritmos: Teoria e Prática)

Material disponibilizado em forma de slides pelo professor Wagner Meira Jr

Instruções para Compilação e Execução

1. Deve-se abrir o terminal.
2. Entrar na pasta TP, usando o comando “cd TP”.
3. Utilizar o comando “make run file=nome” para compilar e executar o código, o comando deve ser exatamente esse, com apenas nome, sendo substituído pelo nome do arquivo, incluindo a extensão, por exemplo, “entrada.txt” é algo que poderia ser colocado para substituir o termo nome no comando dito. Entretanto certifique-se de que o arquivo desejado esteja no diretório atual (exemplo: se a pasta se chama TP e tenha subpastas bin, obj, src e include, o arquivo deve estar na pasta TP, mas não nas subpastas e deve ser da extensão “txt”) do projeto, mas nem em “bin”, nem em “include”, nem em “src” e nem em “obj”.
4. Após avaliação e uso, caso desejado, utilizar o comando “make clean” para deletar os arquivos desnecessários, como o executável e arquivos da pasta “obj” (“.o”).