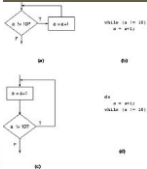


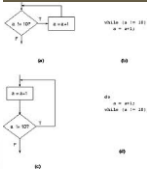
Estruturas de Dados

- Prof. Edkallenn Lima
- edkallenn@yahoo.com.br (somente para dúvidas)
- Blogs:
 - <http://professored.wordpress.com> (Computador de Papel – O conteúdo da forma)
 - <http://professored.tumblr.com/> (Pensamentos Incompletos)
 - <http://umcientistaporquinzena.tumblr.com/> (Um cientista por quinzena)
 - <http://eulinoslivros.tumblr.com/> (Eu Li nos Livros)
 - <http://linabiblia.tumblr.com/> (Eu Li na Bíblia)
- Redes Sociais:
 - <http://www.facebook.com/edkallenn>
 - <http://twitter.com/edkallenn>
 - <https://plus.google.com/u/0/113248995006035389558/posts>
 - [Pinterest: https://www.pinterest.com/edkallenn/](https://www.pinterest.com/edkallenn/)
 - [Instagram: http://instagram.com/edkallenn](https://instagram.com/edkallenn) ou [@edkallenn](https://instagram.com/edkallenn)
 - [LinkedIn: br.linkedin.com/in/Edkallenn](https://br.linkedin.com/in/Edkallenn)
 - [Foursquare: https://pt.foursquare.com/edkallenn](https://pt.foursquare.com/edkallenn)
- Telefones:
 - 68 8401-2103 (VIVO) e 68 3212-1211.
- Os exercícios devem ser enviados SEMPRE para o e-mail: edkevan@gmail.com ou para o e-mail: edkallenn.lima@uninorteac.edu.br



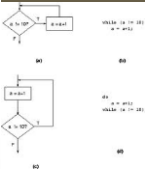
Agenda

- Estruturas (Registros)
- Referenciando elementos
- Declaração de novos tipos
- typedef
- Atribuição
- Operações
- Passagem para funções
- Funções que retornam estruturas
- Arrays de estruturas
- Uniões, Enumerações e tipos
- Estruturas autoreferenciadas



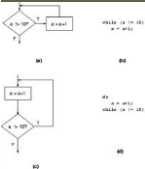
Estruturas de dados heterogêneas (estruturas)

- **Estrutura** é uma coleção de variáveis referenciada por um nome.
- **Estruturas** são tipos de variáveis que agrupam dados geralmente desiguais
- Os itens da estrutura são chamados **membros** ou **campos**
- Algumas linguagens de programação chamam as estruturas de **registros**.
- Uma **definição de estrutura** forma um modelo que pode ser usado para criar variáveis de estruturas



Estruturas

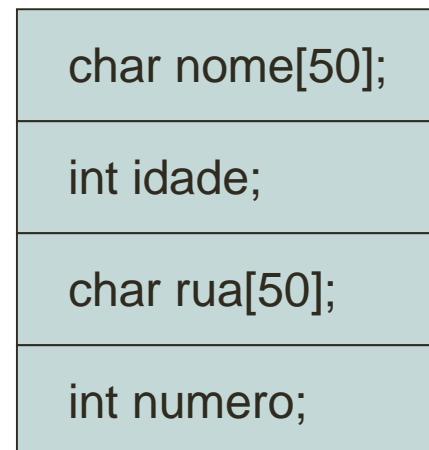
- Geralmente todos os elementos de uma estrutura são **relacionados**
- Um exemplo é um **registro** de folha de pagamento onde um funcionário é descrito por muitos atributos
- A palavra chave **struct** informa ao compilador que um modelo de estrutura está sendo definido.
- Exemplos:



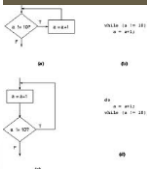
Estruturas

- Uma estrutura pode ser vista como um agrupamento de dados.
- Ex.: cadastro de pessoas.
 - Todas essas informações são da mesma pessoa, logo podemos agrupá-las.
 - Isso facilita também lidar com dados de outras pessoas no mesmo programa

```
struct cadastro{
    char nome[50];
    int idade;
    char rua[50];
    int numero;
};
```



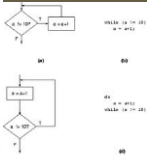
cadastro



Exemplos (tipos derivados)

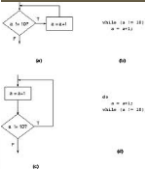
```
struct carta{
    char *face;
    char *naipe;
};
```

```
struct endereco{
    char nome[30];
    char rua[50];
    char cidade[20];
    char estado[2];
    unsigned long int cep;
};
```



Estruturas

- Os **membros** de uma **estrutura** podem ser dos **tipos simples** ou **agregados**, como **matrizes** e outras **estruturas**
- A definição termina com um ponto e vírgula
- A definição é um **comando**;



Tipos estruturados

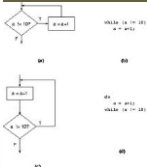
- Qual é a Motivação?
- **Manipulação de dados compostos ou estruturados**
- Exemplos:
 - **ponto no espaço bidimensional**
 - representado por duas coordenadas (x e y), mas tratado como um único objeto (ou tipo)
 - **dados associados a aluno:**
 - aluno representado pelo seu nome, número de matrícula, endereço, etc ., estruturados em um único objeto (ou tipo)

Ponto

X
Y

Aluno

Nome	
Matr	
End	Rua
	No
	Compl



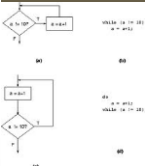
Acessando elementos

- Tipo estrutura:
 - tipo de dado com campos **compostos** de tipos mais **simples (ou estruturados)**
 - elementos acessados através do operador de acesso “ponto” (.)

```

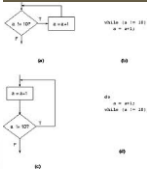
struct ponto                                /* declara ponto do tipo struct */
{
    float x;
    float y;
};
...
struct ponto p;                             /* declara p como variável do tipo struct ponto */
...
p.x = 10.0;                                 /* acessa os elementos de ponto */
p.y = 5.0;

```



Estruturas

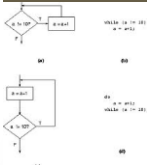
- Após ter sido **definido**, o novo tipo **existe** e pode ser utilizado para **criar variáveis** de modo similar a qualquer tipo simples.
- Exemplo:



```

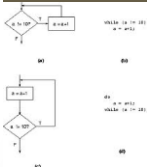
1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função :Primeiro exemplo de estrutura
4     Autor : Edkallenn - Data : 06/04/2012
5  */
6  struct Aluno{
7      int nmat;
8      float nota[3];
9      float media;
10
11 };
12 main(){
13
14     struct Aluno Manoel;    //declara uma variavel
15                             //do tipo Aluno
16
17     Manoel.nmat=456;
18     Manoel.nota[0]=7.5;
19     Manoel.nota[1]=5.2;
20     Manoel.nota[2]=8.4;
21
22     printf("%g\n", Manoel.nota[0]);
23     printf("%g\n", Manoel.nota[1]);
24     printf("%g\n", Manoel.nota[2]);
25
26     Manoel.media =
27     (Manoel.nota[0] + Manoel.nota[1] + Manoel.nota[2]) / 3.0;
28
29     printf("Matricula: %d\n", Manoel.nmat);
30     printf("Media: %.2f\n", Manoel.media);
31
32     getch();
33 }

```



Exercício

- Declare uma estrutura capaz de armazenar o número e 3 notas para um dado aluno.



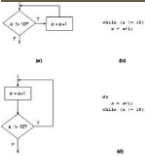
Exercício - Solução

- Possíveis soluções

```
struct aluno {
    int num_aluno;
    int nota1, nota2, nota3;
};
```

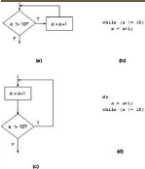
```
struct aluno {
    int num_aluno;
    int nota1;
    int nota2;
    int nota3;
};
```

```
struct aluno {
    int num_aluno;
    int nota[3];
};
```



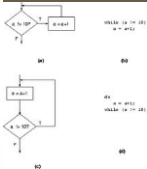
Estrutura

- O escopo de aluno é **global** (antes do main())
- Poderíamos colocar dentro do main ()
restringindo seu **acesso**.



Exercício

- Altere o primeiro programa para que o usuário insira as três notas do aluno Manoel.
- Colocar a estrutura dentro do main



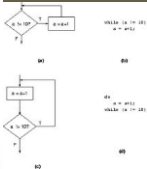


Declarando uma variável

- A instrução

```
struct Aluno Manoel;
```

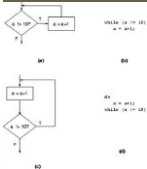
- Declara uma variável de nome **Manoel** do tipo **struct Aluno**.
- Essa declaração reserva espaço suficiente para armazenar todos os membros da estrutura: 4 bytes para **nmat**, 12 bytes para a matriz **nota** e 4 bytes para **media**.



Estruturas

- O uso de estruturas facilita na manipulação dos dados do programa. Imagine declarar 4 cadastros, para 4 pessoas diferentes:

```
char nome1[50], nome2[50], nome3[50], nome4[50];
int idade1, idade2, idade3, idade4;
char rua1[50], rua2[50], rua3[50], rua4[50];
int numero1, numero2, numero3, numero4;
```

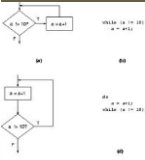


Estruturas

- Utilizando uma estrutura, o mesmo pode ser feito da seguinte maneira:

```
struct cadastro{
    char nome[50];
    int idade;
    char rua[50]
    int numero;
};

//declarando 4 cadastros
struct cadastro c1, c2, c3, c4, c5;
```



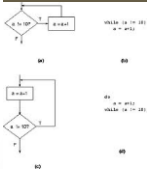
Novos nomes para tipos existentes: typedef

- Declarações com typedef não produzem novos tipos de dados.
- Criam apenas novos nomes (sinônimos ou alias) para os tipos existentes
- Sintaxe:

typedef tipo-existente sinônimo;

- Exemplo:

```
typedef unsigned char BYTE;
typedef unsigned int uint;
int main() {
        BYTE ch;
        uint x;
}
```

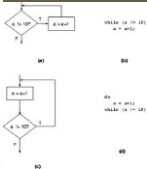


Usando typedef com struct

- Com estruturas podemos utilizar typedef de 3 formas:

```

8      struct Aluno{
9          int nmat;
10         float nota[3];
11         float media;
12     };
13
14     typedef struct Aluno Aluno;
15
16     Aluno Manoel;
    
```







```
#include <stdio.h>
#include <stdlib.h>

/* Função :Primeiro exemplo de estrutura
   Autor : Edkallenn - Data : 06/04/2012
*/

main(){

    typedef struct Aluno{
        int nmat;
        float nota[3];
        float media;
    } Aluno;

    Aluno Manoel;

    Manoel.nmat=456;
    Manoel.nota[0]=7.5;
    Manoel.nota[1]=5.2;
    Manoel.nota[2]=8.4;

    printf("%g\n", Manoel.nota[0]);
    printf("%g\n", Manoel.nota[1]);
    printf("%g\n", Manoel.nota[2]);

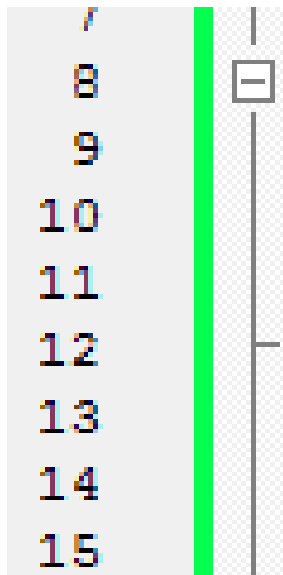
    Manoel.media =
        (Manoel.nota[0] + Manoel.nota[1] + Manoel.nota[2]) / 3.0;

    printf("Matricula: %d\n", Manoel.nmat);
    printf("Media: %.2f\n", Manoel.media);

    getchar();
}
```

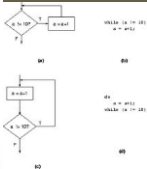
A última forma

- Não informar a etiqueta da estrutura:



```
typedef struct{
    int nmat;
    float nota[3];
    float media;
} Aluno;

Aluno Manoel;
```





Acessando os membros

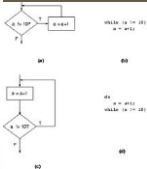
```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct ponto{
5      float x;
6      float y;
7  };
8
9  main() {
10     struct ponto p;
11     printf("\n\nDigite as coordenadas do ponto(x,y): ");
12     scanf("%f,%f", &p.x, &p.y);
13     printf("\n\nO ponto digitado foi: P(%.2f, %.2f)\n", p.x, p.y);
14     getchar();
15 }

```

Basta escrever `&p.x` em lugar de `&(p.x)`.

O operador de acesso ao campo da estrutura tem precedência sobre o operador "endereço de"



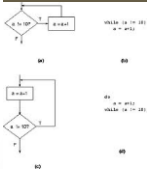
Acessando os membros

- Uma vez criada a estrutura, seus membros podem ser acessados, como vimos por meio do operador ponto.

- A instrução

Manoel.nmat = 456;

- Atribui o valor 456 ao membro nmat da Variável Manoel.
- O **operador ponto** conecta o nome de uma **variável estrutura** a um **membro dela**.
- A Linguagem C trata os membros de uma **estrutura** como quaisquer outras **variáveis simples**.



Combinando declarações

- Podemos definir a estrutura e declarar variáveis na mesma instrução.

- Exemplo:

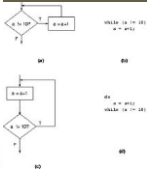

```
struct Aluno{
    int nmat;
    float nota[3];
    float media;
} Manoel, Jose, Ana, Joao;
```

- Se nenhuma outra variável for declarada posteriormente a etiqueta pode ser suprimida.

Assim:

```
struct{
    int nmat;
    float nota[3];
    float media;
} Manoel, Jose, Ana, Joao;
```

- Mais compacta, mas menos clara e flexível.



Inicialização de Estruturas

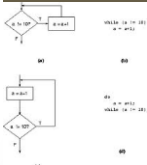
- A inicialização de Estruturas é semelhante a de matrizes.
- Uma variável de estrutura só pode ser inicializada em tempo de compilação; Assim, deve ser das classes extern ou static.
- Exemplo:

```
typedef struct
{
    int dia;
    char mes[10];
    int ano;
```

```
}Data;
```

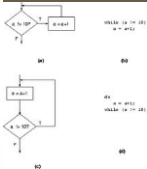
```
static Data natal = {25, "Dezembro", 2001};
static Data aniversario = {07, "Julho", 2012};
```

```
//static sao variaveis permanentes que mantem seus valores
//entre chamadas de funções.
```



Inicialização de Estruturas

- **IMPORTANTE:** Os valores a serem atribuídos a seus membros devem ser colocados na ordem em que foram definidos na estrutura, separados por vírgulas, entre chaves

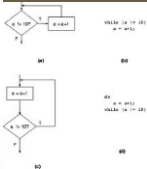


Definindo e inicializando

- Quando NÃO utilizamos a palavra typedef, podemos inicializar e definir na mesma instrução. Assim:

```
struct Data
{
    int dia;
    char mes[10];
    int ano;
}natal = {25, "Dezembro", 2001},
aniversario = {07, "Julho", 2012};

printf("Natal: %d, %s, %d\n", natal.dia, natal.mes, natal.ano);
printf("Niver: %d, %s, %d\n", aniversario.dia,
        aniversario.mes, aniversario.ano);
```

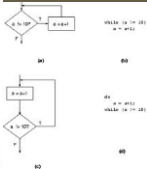


Atribuições entre estruturas

- Uma variável estrutura pode ser atribuída a outra DO MESMO TIPO por meio de uma atribuição simples. Assim:

```
struct Data
{
    int dia;
    char mes[10];
    int ano;
}natal = {25, "Dezembro", 2001},
aniversario1 = {07, "Julho", 2012};

static Data aniversario = {30, "Julho", 2012};
Data Andre;
Andre = aniversario;
printf("Niver Andre: %d, %s, %d\n", Andre.dia, Andre.mes, Andre.ano);
```



Estruturas Aninhadas

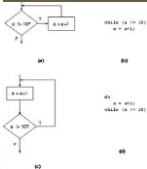
- Assim como podemos ter matrizes onde cada elemento é uma matriz, podemos definir estruturas com membros que sejam outras estruturas.

```
typedef struct
{
    int dia;
    char mes[10];
    int ano;
}Data;

typedef struct
{
    int pecas;
    float preco;
    Data diavenda;
}Venda;

static Venda A = {20, 110.23, {7, "Novembro", 2012}};

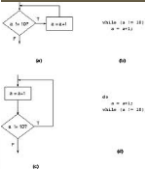
printf("Pecas: %d\n", A.pecas);
printf("Preco: %g\n", A.preco);
printf("Data: %d de %s de %d\n", A.diavenda.dia,
      A.diavenda.mes, A.diavenda.ano);
```



Ponteiros para estruturas

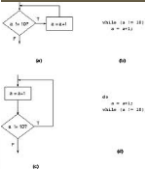
- Os parênteses são indispensáveis, no caso anterior, porque o operador “conteúdo de” tem precedência menor do que o operador de acesso ponto
- O acesso a campos de estrutura é tão comum em programas C que a linguagem oferece outro operador de acesso, que permite acessar campos a partir do ponteiro da estrutura
- Esse operador é composto por um traço seguido de um sinal de maior, formando uma seta (->)
- Portanto, podemos reescrever a atribuição anterior da seguinte forma:

```
pp->x = 12.0;
```



Ponteiros para estruturas

- Resumindo, se temos uma variável estrutura e queremos acessar seus campos, usamos o operador de acesso ponto **(p.x)**;
- Se temos uma variável ponteiro para estrutura, usamos o operador de acesso seta **(pp->x)**.
- Seguindo este raciocínio, se temos o ponteiro e queremos acessar o endereço de um campo, fazemos **&pp->x**.



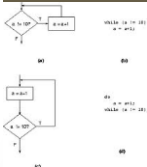
Ponteiros para estruturas - Resumo

- acesso ao valor de um campo x de uma variável estrutura p: `p.x`
- acesso ao *valor* de um campo x de uma variável ponteiro pp: `pp->x`
- acesso ao *endereço* do campo x de uma variável ponteiro pp: `&pp->x`

```
struct ponto *pp;
```

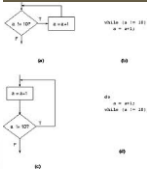
```
(*pp).x = 12.0;      /* formas equivalentes de acessar o valor de um campo x */
```

```
pp->x = 12.0;
```



Passando estruturas para funções

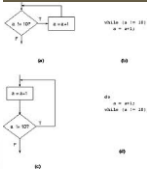
- Estruturas podem ser passadas para funções da mesma forma que variáveis simples
- O nome de uma **estrutura NÃO é um endereço**, portanto ela pode ser passada **por valor**
- A função **recebe toda a estrutura** como parâmetro
- A função **acessa a cópia da estrutura** na pilha
- função **não altera** os valores dos campos da estrutura original
- A operação **pode ser custosa** se a estrutura for muito grande



Passando estruturas para funções

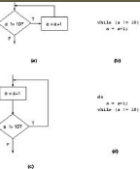
- Exemplo:

```
/* função que imprima as coordenadas do ponto */
void imprime (struct ponto p)
{
    printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
}
```



Passando estruturas para funções

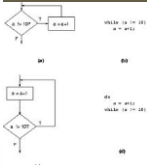
- Por valor: próximo slide
- É feita uma CÓPIA DA ESTRUTURA



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Captura e imprime um ponto
4     Autor : Edkallenn - Data : 02/06/2016
5     Obs: Usando passagem por valor */
6  typedef struct ponto{
7     char letra;
8     float x;
9     float y;
10 }Ponto;
11
12 void imprime(Ponto);
13 Ponto captura();
14
15 main(){
16     Ponto p;
17     p = captura();
18     imprime(p);
19     getchar();
20 }
21 void imprime(Ponto p){
22     printf("O ponto fornecido foi: (%.2f, %.2f)\n", p.x, p.y);
23 }
24 Ponto captura(){
25     Ponto b;
26     printf("Digite as coordenadas do ponto(x,y): ");
27     scanf("%f,%f", &b.x, &b.y);
28     return b;
29 }

```



Passagem para funções por referência

- Apenas o ponteiro da estrutura é passado, mesmo que não seja necessário alterar os valores dos campos dentro da função

```

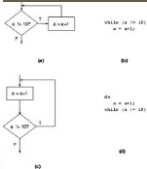
1  #include <stdlib.h>
2  #include <stdio.h>
3  struct ponto{
4      float x;
5      float y;
6  };
7  void imprime(struct ponto*);
8  void captura(struct ponto*);
9  main(){
10     struct ponto p;
11     captura(&p);
12     imprime(&p);
13     getch();
14 }
15 void imprime(struct ponto* pp){
16     printf("O ponto fornecido foi: (%.2f, %.2f)\n", pp->x, pp->y);
17 }
18 void captura(struct ponto* pp){
19     printf("Digite as coordenadas do ponto(x,y): ");
20     scanf("%f %f", &pp->x, &pp->y);
21 }

```



Passando estruturas inteiras para funções

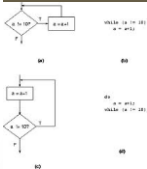
- Quando uma estrutura inteira é usada como parâmetro para uma função, a estrutura inteira é passada usando o método padrão de chamada **por valor**
- Isso implica que quaisquer alterações podem ser feitas no conteúdo da estrutura dentro da função para a qual ela é passada **sem afetar** a estrutura usada como argumento.





Cuidados a se tomar

- Se for declarado parâmetros que são estruturas inteiras, dever-se-á **tornar a estrutura global**, para que **todas as partes** do programa possam usá-la.
- Se, no exemplo anterior, **struct_exemplo** tivesse sido declarada dentro de **main()**, ela não seria visível a **funcao1()**
- O tipo de argumento deve coincidir com o tipo de parâmetro. Não é suficiente que sejam fisicamente semelhantes, os nomes dos tipos devem também coincidir.



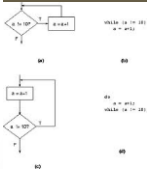


```

40  [- Venda TotalVendas(Venda C, Venda D) {
41      Venda T;
42      T.pecas = C.pecas + D.pecas;
43      T.preco = C.preco + D.preco;
44      return T;
45  }
46

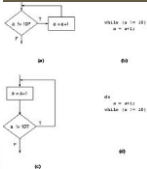
```

- O protótipo da função TotalVendas() e as instruções de seu corpo tratam as variáveis argumentos como se trata qualquer tipo simples.
- A função cria as novas variáveis C e D para conter cópia dos valores das variáveis A e B de main(), enviadas como argumento, como qualquer outra variável simples passada por valor.
- Então as variáveis C e D não são as mesmas das de main()
- A função poderia alterá-las sem contudo alterar as originais.



Arrays de Estruturas

- Uma lista de peças e preços, como no exemplo anterior, é composta por várias vendas (provavelmente mais de duas)
- Cada venda é descrita por uma variável do tipo **Venda**.
- Para tratar de várias vendas, é perfeitamente correto pensar em um Array de estruturas.
- O próximo programa demonstra o uso de uma matriz de estruturas



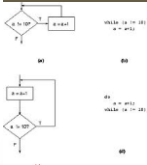




```

57 void novavenda() {
58     printf("Dia: "); scanf("%d", &vendas[n].diavenda.dia);
59     printf("Mes: "); scanf("%s", vendas[n].diavenda.mes);
60     printf("Ano: "); scanf("%d", &vendas[n].diavenda.ano);
61     printf("\n");
62     printf("Pecas: "); scanf("%d", &vendas[n].pecas);
63     printf("Preco: "); scanf("%f", &vendas[n].preco);
64     Total.pecas += vendas[n].pecas;
65     Total.preco += vendas[n++].preco;
66 }
67 void listavenda() {
68     int i;
69     if(!n) {
70         puts("Lista Vazia");
71         return;
72     }
73     printf("\n\nRelatorio\n");
74     for(i=0; i<n; i++){
75         printf("%2d de %10s de %4d", vendas[i].diavenda.dia,
76             vendas[i].diavenda.mes, vendas[i].diavenda.ano);
77         printf("\nPecas: %10d\t", vendas[i].pecas);
78         printf("Preco: %20.2f\n", vendas[i].preco);
79     }
80     printf("\nTotal");
81     printf("%29d", Total.pecas);
82     printf("%20.2f\n\n", Total.preco);
83 }
84

```

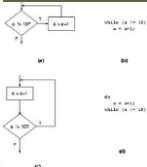


Matriz de estruturas

- Declarar uma matriz de estruturas é exatamente igual a declarar qualquer outra matriz.

```
Venda vendas[50];
```

- Esta instrução declara **vendas** como sendo um vetor de 50 elementos. Cada elemento da matriz é uma estrutura do tipo **Venda**.
- Logo, **vendas[0]** é a primeira estrutura do tipo **Venda**, **vendas[1]** é a segunda e assim por diante.
- O nome **vendas** é o de uma matriz (portanto um endereço) em que os elementos são estruturas.
- O compilador providencia espaço contínuo de memória para armazenar até 50 estruturas do tipo **Venda**.

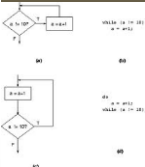


Acessando membros de uma matriz de estruturas

- Membros individuais são acessados aplicando-se o operador ponto seguido do nome da variável, que, nesse caso, é um elemento da matriz.

```
vendas[n].preco
```

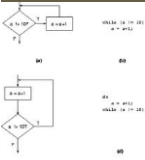
- O subscrito é associado à **vendas** e não ao membro. A expressão acima refere-se ao membro **preco** da n-ésima estrutura da matriz.



Exercício

- Utilizando a estrutura abaixo, faça um programa para ler o número e as 3 notas de 10 alunos.

```
struct aluno {
    int num_aluno;
    float nota1, nota2, nota3;
    float media;
};
```

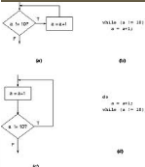


Exercício - Solução

- Utilizando a estrutura abaixo, faça um programa para ler o número e as 3 notas de 10 alunos

```
struct aluno {
    int num_aluno;
    float nota1, nota2, nota3;
    float media;
};
```

```
int main() {
    struct aluno a[10];
    int i;
    for(i=0; i<10; i++) {
        scanf("%d", &a[i].num_aluno);
        scanf("%f", &a[i].nota1);
        scanf("%f", &a[i].nota2);
        scanf("%f", &a[i].nota3);
        a[i].media = (a[i].nota1 + a[i].nota2 + a[i].nota3)/3.0;
    }
}
```

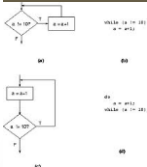


Alocação dinâmica de estruturas

- Tamanho do espaço de memória alocado dinamicamente é dado pelo operador **sizeof** aplicado sobre o tipo estrutura
- Função **malloc** retorna o endereço do espaço alocado, que é então convertido para o tipo ponteiro da estrutura

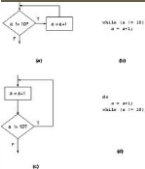
```
struct ponto* p;
p = (struct ponto*) malloc (sizeof(struct ponto));

...
p->x = 12.0;
...
```



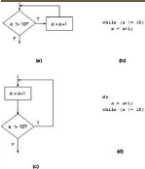
Alocação dinâmica

- No fragmento de código anterior foi alocado, de modo dinâmico, uma única estrutura e o endereço da área alocada foi armazenado em p
- O tamanho do espaço alocado é dado pelo operador sizeof aplicado sobre a estrutura
- Acessa-se normalmente os campos da estrutura com a variável ponteiro que armazena seu endereço.



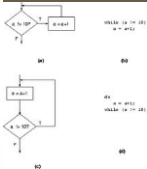
Exercício (em sala)

- Um posto de saúde deseja cadastrar os seguintes dados sobre as pessoas atendidas: **nome, idade, peso e altura**.
- Defina uma estrutura de dados conveniente para armazenar estes dados.
- Considere que o cadastro será armazenado em um vetor.
- O tamanho do vetor deve ser definido dinamicamente (alocação dinâmica)



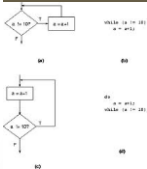
Exercício (fazer em sala!)

- Definir uma estrutura chamada **PerfilSaude** que contenha os seguintes campos:
 - Nome Completo**: string com 80 caracteres
 - Sexo**: um caractere
 - Data de nascimento**: dia, mês e ano separados (DICA: use outra estrutura)
 - Altura**: um número real
 - Peso**: um número real
- Faça um programa que tenha uma função que **receba** esses dados e os **utilize** para definir os membros de uma variável **PerfilSaude**
- O programa deve incluir funções que **calculem** e **retornem** a **idade atual** do usuário em anos, a **frequência cardíaca máxima** e a **frequência cardíaca ideal**, o **índice de Massa Corporal**
- O programa deverá pedir a informação da pessoa, criar uma variável do tipo **PerfilSaude** para ela e exibir as informações dessa variável – o que inclui **TODOS** os dados.
- Em seguida deverá **calcular** e **exibir** a **idade** (em **anos**), seu **IMC**, e suas **frequências cardíacas máxima** e **ideal**. Deverá também exibir a tabela de valores do IMC



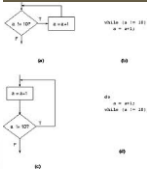
Exercício (fazer em casa!)

- Definir uma estrutura chamada **PerfilSaude** que contenha os seguintes campos:
 - Nome Completo**: string com 80 caracteres
 - Sexo**: um caractere
 - Data de nascimento**: dia, mês e ano separados (DICA: use outra estrutura)
 - Altura**: um número real
 - Peso**: um número real
- Faça um programa que tenha uma função que **receba** esses dados e os **utilize** para definir os membros de uma variável **PerfilSaude**
- O programa deve incluir funções que **calculam** e **retornem** a **idade atual** do usuário em anos, a **frequência cardíaca máxima** e a **frequência cardíaca ideal**, o **índice de Massa Corporal**
- O programa deverá pedir a informação da pessoa, criar uma variável do tipo **PerfilSaude** para ela e exibir as informações dessa variável – o que inclui **TODOS** os dados.
- Em seguida deverá **calcular** e **exibir** a **idade** (em **anos**), seu **IMC**, e suas **frequências cardíacas máxima** e **ideal**. Deverá também exibir a tabela de valores do IMC
- Fazer um vetor de **N** (informado pelo usuário) elementos com as informações do exercício acima. Incluir o vetor nas chamadas de funções. (Usar alocação dinâmica)
- PARA QUARTA-FEIRA (14/06/2017).
- Título do e-mail: [TRAB-ED-Perfil_Saude]NomeSobrenome**
- Nome do arquivo: Perfil_Saude.c**



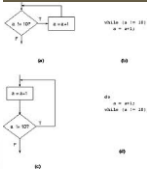
Vetores de Ponteiros para estruturas

- Da mesma forma que podemos declarar vetores de estruturas, podemos também declarar vetores de ponteiros para estruturas
- É útil para economizar memória e para tratar um conjunto de elementos complexos.
- Como exemplo vejamos um problema em que desejamos armazenar uma tabela com diversos dados de alunos.



Vetores de ponteiros para estruturas

- tabela com dados de alunos, organizada em um vetor
- dados de cada aluno:
 - matrícula: número inteiro
 - nome: cadeia com até 80 caracteres
 - endereço: cadeia com até 120 caracteres
 - telefone: cadeia com até 20 caracteres



Vetores de ponteiros para estruturas

- Solução 1:

- Aluno

- estrutura ocupando pelo menos $4+81+121+21 = 227$ Bytes

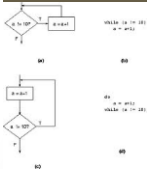
- tab

- vetor de Aluno
 - representa um desperdício significativo de memória, se o número de alunos bem inferior ao máximo estimado

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define MAX 3
5
6  typedef struct aluno{
7      int nmat;
8      char nome[81];
9      char end[121];
10     char tel[20];
11 }Aluno;
12
13 void inicializa(int n, Aluno** tab);
14 void preenche(int n, Aluno** tab, int i);
15 void imprime(int n, Aluno** tab, int i);
16 void imprime_tudo(int n, Aluno** tab);
17 void retira(int n, Aluno** tab, int i);
18

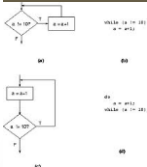
```



- Solução 2 (usada no que se segue):
 - **tab**
 - vetor de ponteiros para **Aluno**
 - elemento do vetor ocupa espaço de um ponteiro
 - alocação dos dados de um aluno no vetor:
 - nova cópia da estrutura **Aluno** é alocada dinamicamente
 - endereço da cópia é armazenada no vetor de ponteiros
 - posição vazia do vetor: **valor é o ponteiro nulo**

21

Aluno* tab[MAX];

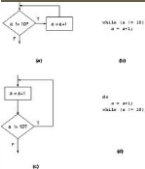


- Inicializa - função para inicializar a tabela:
 - recebe um vetor de ponteiros
 - (parâmetro deve ser do tipo “ponteiro para ponteiro”)
 - atribui NULL a todos os elementos da tabela

```

36 void inicializa(int n, Aluno** tab){
37     int i;
38     for(i=0;i<n;i++){
39         tab[i]=NULL;
40     }
41 }

```

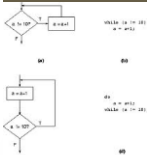


- Preenche - função para armazenar novo aluno na tabela:
 - recebe a posição onde os dados serão armazenados
 - dados são fornecidos via teclado
 - se a posição da tabela estiver vazia, função aloca nova estrutura
 - caso contrário, função atualiza a estrutura já apontada pelo ponteiro

```

43 void preenche(int n, Aluno** tab, int i){
44     if(i<0||i>=n){
45         printf("Índice fora do índice do vetor - preenche\n");
46         exit(1);
47     }
48     if(tab[i] == NULL) {
49         tab[i] = (Aluno*)malloc(sizeof(Aluno));
50         printf("Entre com a matrícula: ");scanf(" %d", &tab[i]->nmat);
51         printf("Entre com o nome: ");      scanf(" %80[^\n]", tab[i]->nome);
52         printf("Entre com o sobrenome: ");scanf(" %120[^\n]", tab[i]->end);
53         printf("Entre com o telefone: "); scanf(" %20[^\n]", tab[i]->tel);
54     }
55 }

```

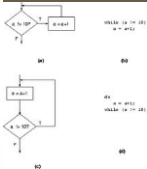


- Retira - função para remover os dados de um aluno da tabela:
 - recebe a posição da tabela a ser liberada
 - libera espaço de memória utilizado para os dados do aluno

```

56 void retira(int n, Aluno** tab, int i){
57     if(i<0||i>=n){
58         printf("Índice fora do índice do vetor - preenche\n");
59         exit(1);
60     }
61     if(tab[i] != NULL) {
62         free(tab[i]);
63         tab[i]=NULL;
64     }
65 }

```

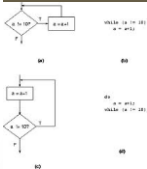


- Imprime - função para imprimir os dados de um aluno da tabela:
 - recebe a posição da tabela a ser impressa

```

67 void imprime(int n, Aluno** tab, int i){
68     if(i<0||i>=n){
69         printf("Índice fora do índice do vetor\n");
70         exit(1);
71     }
72     if(tab[i] != NULL){
73         printf("Matrícula: %d\n", tab[i]->nmat);
74         printf("Nome: %s\n", tab[i]->nome);
75         printf("Sobrenome: %s\n", tab[i]->end);
76         printf("Telefone: %s\n", tab[i]->tel);
77     }
78 }

```

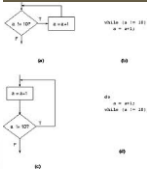


- Imprime_tudo - função para imprimir todos os dados da tabela:
 - recebe o tamanho da tabela e a própria tabela

```

80 void imprime_tudo(int n, Aluno** tab){
81     int i;
82     for(i=0;i<n;i++){
83         imprime(n,tab,i);
84     }
85 }
86

```

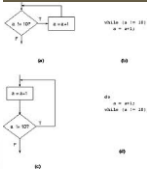


Programa de Teste

```

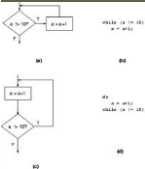
19  int main(){
20      Aluno* tab[MAX];
21      inicializa(MAX,tab);
22      preenche(MAX,tab,0);
23      preenche(MAX,tab,1);
24      preenche(MAX,tab,2);
25      imprime_tudo(MAX, tab);
26      retira(MAX,tab,0);
27      retira(MAX,tab,1);
28      imprime_tudo(MAX, tab);
29      //finaliza saindo deste programa
30      getchar();
31      return 0;
32  }

```



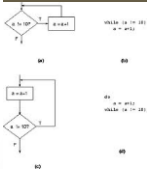
Exercício

- Alterar o programa anterior para que o usuário digite os dados e escolha as opções (Retirar, Preencher, Listar um e todos)
- Usar um menu com as opções
- Tratar as opções e entradas inválidas



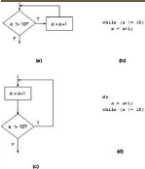
TAD

- Com a implementação **desacoplada** a manutenção é facilitada e o potencial de reuso é aumentado.
- A ideia é deixar os módulos e compilação em separado.
- Vamos ver o exemplo de do tipo de dado ponto
- E em seguida frações (em um único arquivo - para melhor compreensão)



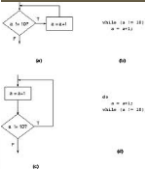
Módulos e Compilação em Separado

- Módulo → um arquivo com funções que representam apenas parte da implementação de um programa completo
- Arquivo objeto → resultado de compilar um módulo – geralmente com extensão *.o* ou *.obj*
- Ligador (linker) → junta todos os arquivos objeto em um único arquivo executável



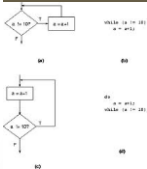
Módulos e Compilação em Separado

- Interface de um módulo de funções:
- É um arquivo contendo apenas:
 - os protótipos das funções oferecidas pelo módulo
 - os tipos de dados exportados pelo módulo (typedef's, struct's, etc)
- em geral possui:
 - nome: igual ao do módulo ao qual está associado
 - E a extensão: *.h*



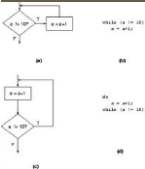
Módulos e Compilação em Separado

- Inclusão de arquivos de interface no código:
- `#include <arquivo.h>`
 - protótipos das funções da biblioteca padrão de C
- `#include "arquivo.h"`
 - protótipos de módulos do usuário (apenas os protótipos)
- Observar um arquivo .h da biblioteca padrão do C



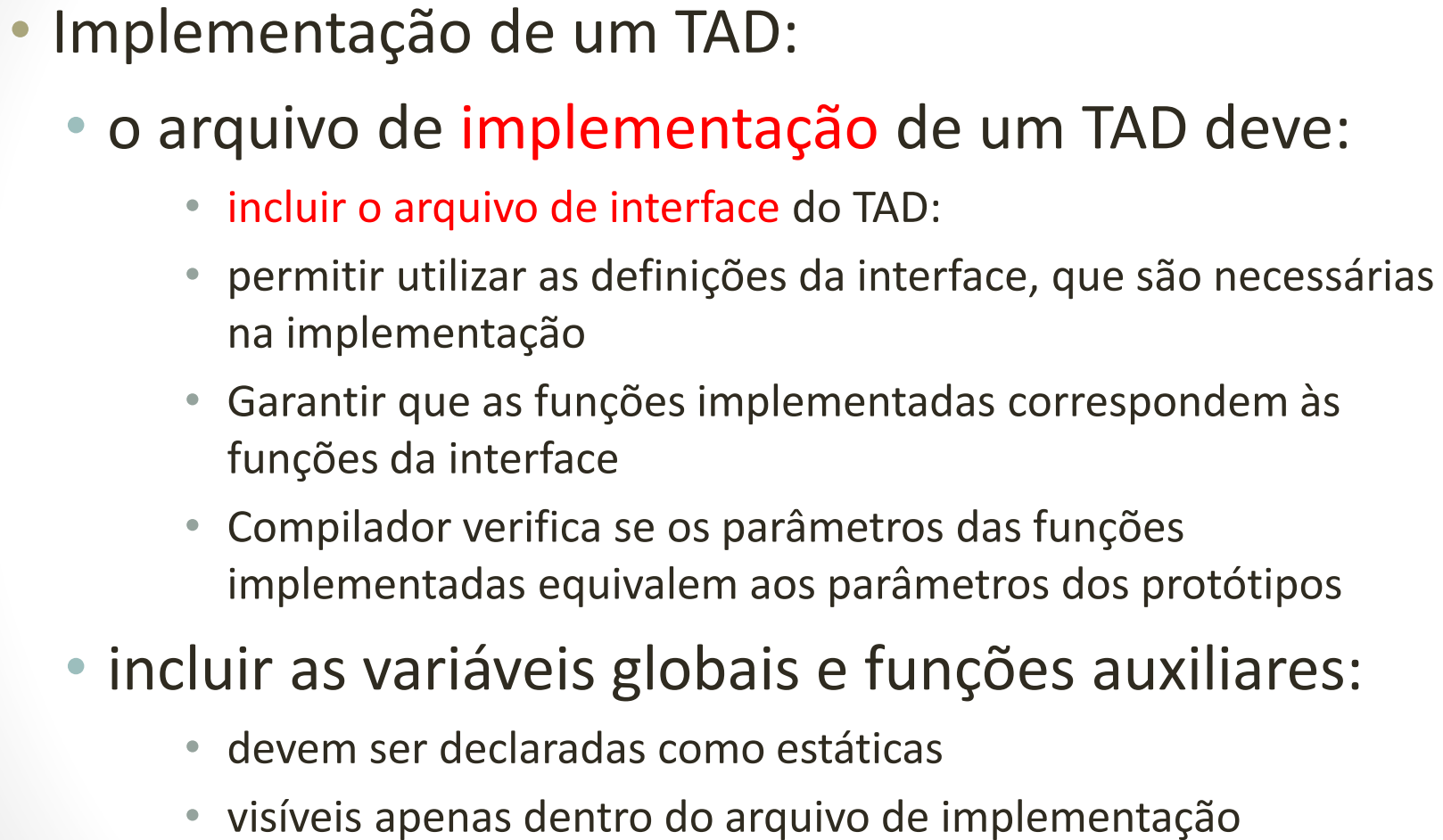
Tipo Abstrato de Dados

- Tipo Abstrato de Dados (TAD):
 - um TAD define:
 - um novo tipo de dado
 - o conjunto de operações para manipular dados desse tipo
 - um TAD facilita:
 - a manutenção e a reutilização de código
 - abstrato = “forma de implementação não precisa ser conhecida”
 - para utilizar um TAD é necessário conhecer a sua **funcionalidade**, mas não a sua **implementação**



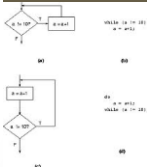
TAD

TAD



Exemplo em um único arquivo

- TAD Fracao
- Define, obtém, soma, subtrai, multiplica, e divide frações.

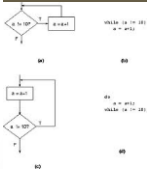


Primeira parte (linhas 1 a 23)

```

1  /* Função : Exemplo de TAD números racionais
2  / Autor : Edkallenn
3  / Data : 02/06/2012
4  / Observações: Mostra a construção de um TAD de numeros
5  / racionais que representa frações
6  */
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <conio.h>
10
11 typedef struct frac{
12     int num;
13     int den;
14 }frac;
15
16 //prototipos
17 void obter_fracoes( frac *, frac *);
18 void somar_fracoes(frac, frac);
19 void subtrair_fracoes(frac, frac);
20 void multiplicar_fracoes(frac, frac);
21 void dividir_fracoes(frac, frac);
22 void simplificar_fracao(frac);
23 int mdc(int x, int y);

```



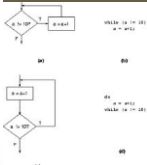
	<pre> while (a < 10) a = a + 1; </pre>
(a)	(b)
	<pre> do a = a + 1; while (a < 10); </pre>
(c)	(d)

Terceira parte (linhas 50 a 81)

```

50 void obter_fracoas(frac *a, frac *b){
51     printf("\nDigite o Numerador da 1a. Fracao: ");
52     scanf("%d", &((*a).num));
53     printf("\nDigite o Denominador da 1a. Fracao: ");
54     scanf("%d", &((*a).den));
55     printf("\nDigite o Numerador da 2a. Fracao: ");
56     scanf("%d", &((*b).num));
57     printf("\nDigite o Denominador da 2a. Fracao: ");
58     scanf("%d", &((*b).den));
59     printf("\nA fracao 1 e: %d / %d\n", (*a).num, (*a).den);
60     printf("A fracao 2 e: %d / %d\n", (*b).num, (*b).den);
61     system("pause");
62     return;
63 }
64 void somar_fracoas(frac x, frac y){
65     frac f;
66     f.num = x.num*y.den + y.num*x.den;
67     f.den = x.den*y.den;
68     simplificar_fracao(f);
69 }
70 void subtrair_fracoas(frac x, frac y){
71     frac f;
72     f.num = x.num*y.den - y.num*x.den;
73     f.den = x.den*y.den;
74     simplificar_fracao(f);
75 }
76 void multiplicar_fracoas(frac x, frac y){
77     frac f;
78     f.num = x.num*y.num;
79     f.den = x.den*y.den;
80     simplificar_fracao(f);
81 }

```

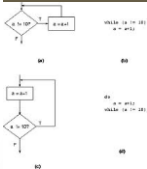


Quarta parte (linhas 82-105)

```

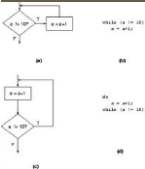
82  void dividir_fracoes(frac x, frac y){
83      frac f;
84      f.num = x.num*y.den;
85      f.den = x.den*y.num;
86      simplificar_fracao(f);
87  }
88  void simplificar_fracao(frac f){
89      int m;
90      m = mdc(f.num, f.den);
91      f.num = f.num/m;
92      f.den = f.den/m;
93      printf("Resultado: %d / %d\n", f.num, f.den);
94      system("pause");
95  }
96  int mdc(int x, int y){
97      int r;
98      while(r!=0){
99          r = x % y;
100         x = y;
101         y = r;
102     }
103     return x;
104 }
105

```



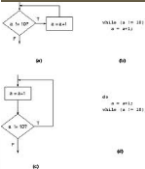
Exercício

- Modifique a funcao simplificar_fracao do programa anterior para, em vez de exibir, esta função retorne como valor a fração simplificada
- Crie um TAD Complexo para realizar aritmética com números complexos. Utilize variáveis double para representar os campos deste tipo. Implemente funções para ler um número complexo, somar dois números complexos, subtrair dois números complexos, mostrar um número complexo na forma (a,b) onde a é a parte real e b, a parte imaginária).



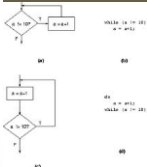
Exercícios

- Crie um tipo Data para tratar dados representando os valores de dia, mes e ano. Implemente funções para ler uma data válida e para mostra a data no formato dd/mm/aaaa. Implemente as funções DiaSeguinte e DiaAnterior para incrementar e decrementar a data, respectivamente, mantendo-a válida.
- Crie um TAD chamado Retangulo com campos comprimento e largura. Crie funções para calcular o perimetro e a area de um retangulos gerados aleatoriamente




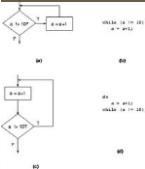
Tipo Abstrato de Dados

- TAD Ponto
 - tipo de dado para representar um ponto no R2 com as seguintes operações:
 - **cria** cria um ponto com coordenadas x e y
 - **libera** libera a memória alocada por um ponto
 - **acessa** retorna as coordenadas de um ponto
 - **atribui** atribui novos valores às coordenadas de um ponto
 - **distancia** calcula a distância entre dois pontos



Tipo Abstrato de Dados

- Interface de Ponto
 - Define o nome do tipo e os nomes das funções exportadas
 - A composição da estrutura Ponto não faz parte da interface:
 - não é exportada pelo módulo
 - não faz parte da interface do módulo
 - não é visível para outros módulos
- 
- Os módulos que utilizarem o TAD Ponto:
 - não poderão acessar diretamente os campos da estrutura Ponto
 - só terão acesso aos dados obtidos através das funções exportadas

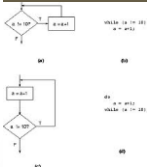


• Arquivo *ponto.h* – com a interface de *Ponto*

```

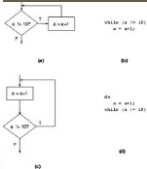
1  /* TAD: Ponto (x,y) */
2  /* Tipo exportado */
3  typedef struct ponto Ponto;
4
5  /* Funções exportadas */
6  /* Função cria - Aloca e retorna um ponto com coordenadas (x,y) */
7  Ponto* pto_cria (float x, float y);
8
9  /* Função libera - Libera a memória de um ponto previamente criado */
10 void pto_libera (Ponto* p);
11
12 /* Função acessa - Retorna os valores das coordenadas de um ponto */
13 void pto_acessa (Ponto* p, float* x, float* y);
14
15 /* Função atribui - Atribui novos valores às coordenadas de um ponto */
16 void pto_atribui (Ponto* p, float x, float y);
17
18 /* Função distancia - Retorna a distância entre dois pontos */
19 float pto_distancia (Ponto* p1, Ponto* p2);
20

```



Tipo Abstrato de Dados

- Implementação de Ponto:
 - inclui o arquivo de interface de Ponto
 - define a composição da estrutura Ponto
 - inclui a implementação das funções externas



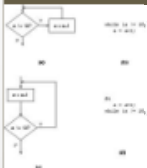
- *ponto.c* – arquivo com o TAD ponto

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h>
4  #include "ponto.h"
5
6  struct ponto {
7      float x;
8      float y;
9  };
10
11  Ponto *pto_cria (float x, float y) ...
12
13  void pto_libera (Ponto *p) ...
14
15  void pto_acessa (Ponto *p, float *x, float *y) ...
16
17  void pto_atribui (Ponto *p, float x, float y) ...
18
19  float pto_distancia (Ponto *p1, Ponto *p2) ...
20
21  // Veja os próximos slides para incluir cada função
22  // aqui em seu devido lugar
23
24

```

- Ver os próximos slides para implementação das funções externas

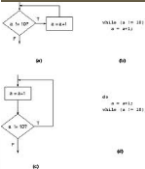


- Função para criar um ponto dinamicamente:
- aloca a estrutura que representa o ponto
- inicializa os seus campos

```

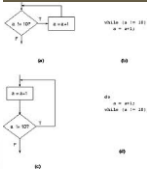
Ponto* pto_cria (float x, float y){
    Ponto* p = (Ponto*) malloc(sizeof(Ponto));
    if (p == NULL) {
        printf("Memória insuficiente!\n");
        exit(1);
    }
    p->x = x;
    p->y = y;
    return p;
}

```



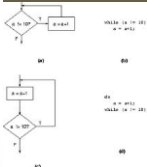
- Função para liberar um ponto:
- deve apenas liberar a estrutura criada dinamicamente através da função *cria*

```
void pto_libera (Ponto* p)
{
    free(p);
}
```



- Função para calcular a distância entre dois pontos

```
float pto_distancia (Ponto* p1, Ponto* p2)
{
    float dx = p2->x - p1->x;
    float dy = p2->y - p1->y;
    return sqrt(dx*dx + dy*dy);
}
```

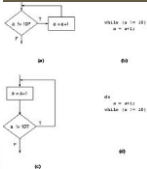


```

11 Ponto *pto_cria (float x, float y) {
12     Ponto *p = (Ponto*) malloc(sizeof(Ponto));
13     if (p == NULL) {
14         printf("Memoria insuficiente!\n");
15         exit(1);
16     }
17     p->x = x;
18     p->y = y;
19     return p;
20 }
21
22 void pto_libera (Ponto *p)
23 {
24     free(p);
25 }
26
27 void pto_acessa (Ponto *p, float *x, float *y)
28 {
29     *x = p->x;
30     *y = p->y;
31 }
32
33 void pto_atribui (Ponto *p, float x, float y)
34 {
35     p->x = x;
36     p->y = y;
37 }
38
39 float pto_distancia (Ponto *p1, Ponto *p2)
40 {
41     float dx = p2->x - p1->x;
42     float dy = p2->y - p1->y;
43     return sqrt(dx*dx + dy*dy);
44 }

```

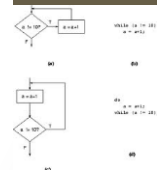
- Aqui estão
Somente as
funções (de
ponto.c)
- Seu arquivo
deve estar
assim!
- Lembrando
que este é a
continuação
do arquivo
ponto.c



Exemplo de arquivo que usa o TAD Ponto

estruturas\ponto.h estruturas\testa_ponto.c estruturas\ponto.c

```
1  /* Função : Exemplo de uso do TAD Ponto
2   / Autor : Edkallenn
3   / Data : 06/04/2012
4   / Observações: Salve este arquivo como testa_ponto.c
5   */
6   #include <stdio.h>
7   #include <math.h>
8   #include "ponto.h"
9
10
11  int main (void)
12  {
13      //float x, y;
14
15      Ponto *p = pto_cria(2.0,1.0);
16      Ponto *q = pto_cria(3.4,2.1);
17
18      float d = pto_distancia(p,q);
19
20      printf("Distancia entre pontos: %f\n",d);
21
22      pto_libera(q);
23      pto_libera(p);
24
25      return 0;
26  }
```



Uso do programa com TAD Ponto

estruturas\ponto.c

estruturas\testa_ponto.c x

estruturas\ponto.h

```

1  /* Função : Exemplo de uso do TAD Ponto
2  / Autor : Edkallenn
3  / Data : 06/04/2012
4  / Observações: Salve este arquivo como testa_ponto.c
5  */
6  #include <stdio.h>
7  #include <math.h>
8  #include "ponto.h"
9
10
11 int main (void)
12 {
13     //float x, y;
14
15     Ponto *p = pto_cria(2.0,1.0);
16     Ponto *q = pto_cria(3.4,2.1);
17
18     float d = pto_distancia(p,q);
19
20     printf("Distancia entre pontos: %f\n",d);
21
22     pto_libera(q);
23     pto_libera(p);
24
25     return 0;
26 }
27

```

"C:\Users\Ed\Dropbox\001 - Estrutura de dados - ed\programas-c\TAD\TAD_ponto\bin\Debug\TAD_ponto.exe"

Distancia entre pontos: 1.780449

Process returned 0 (0x0) execution time : 0.146 s

Press any key to continue.

logs & others

Code::Blocks

Search results

Cccc

Build log x

Build messages

CppCheck

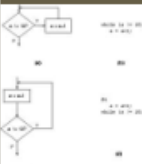
CppCheck messages

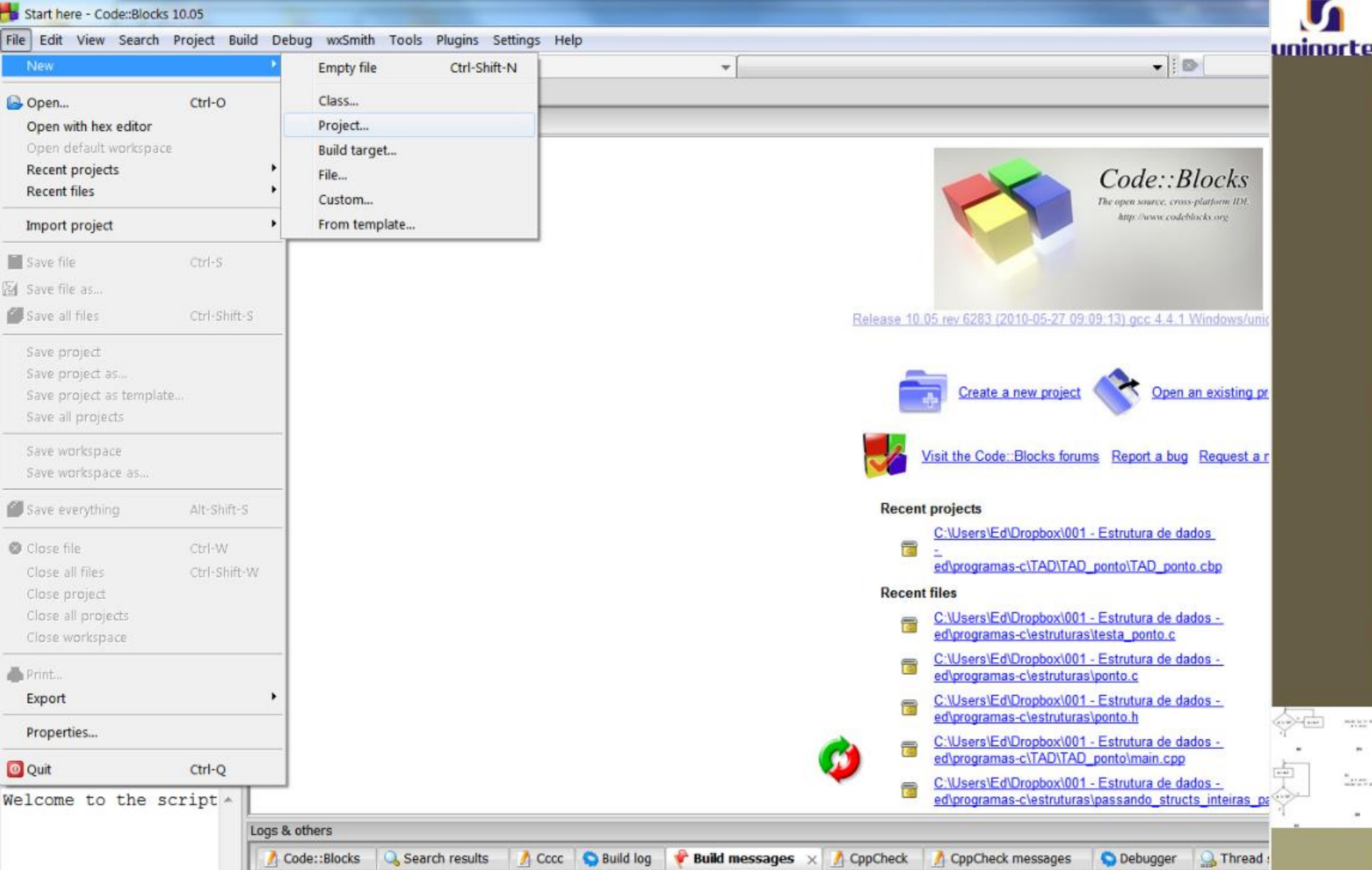
Debugger

Thread search

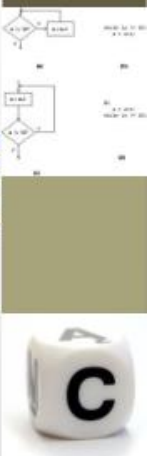
Checking for existence: C:\Users\Ed\Dropbox\001 - Estrutura de dados - ed\programas-c\TAD\TAD_ponto\bin\Debug\TAD_ponto.exe

Executing: "C:\Program Files (x86)\CodeBlocks\cb_console_runner.exe" "C:\Users\Ed\Dropbox\001 - Estrutura de dados - ed\programas-c\TAD\TAD_ponto\bin\Debug\TAD_ponto.exe" (in C:\Users\Ed\Dropbox\001 - Estrutura de dados - ed\programas-c\TAD\TAD_ponto\.)





File → New → Project

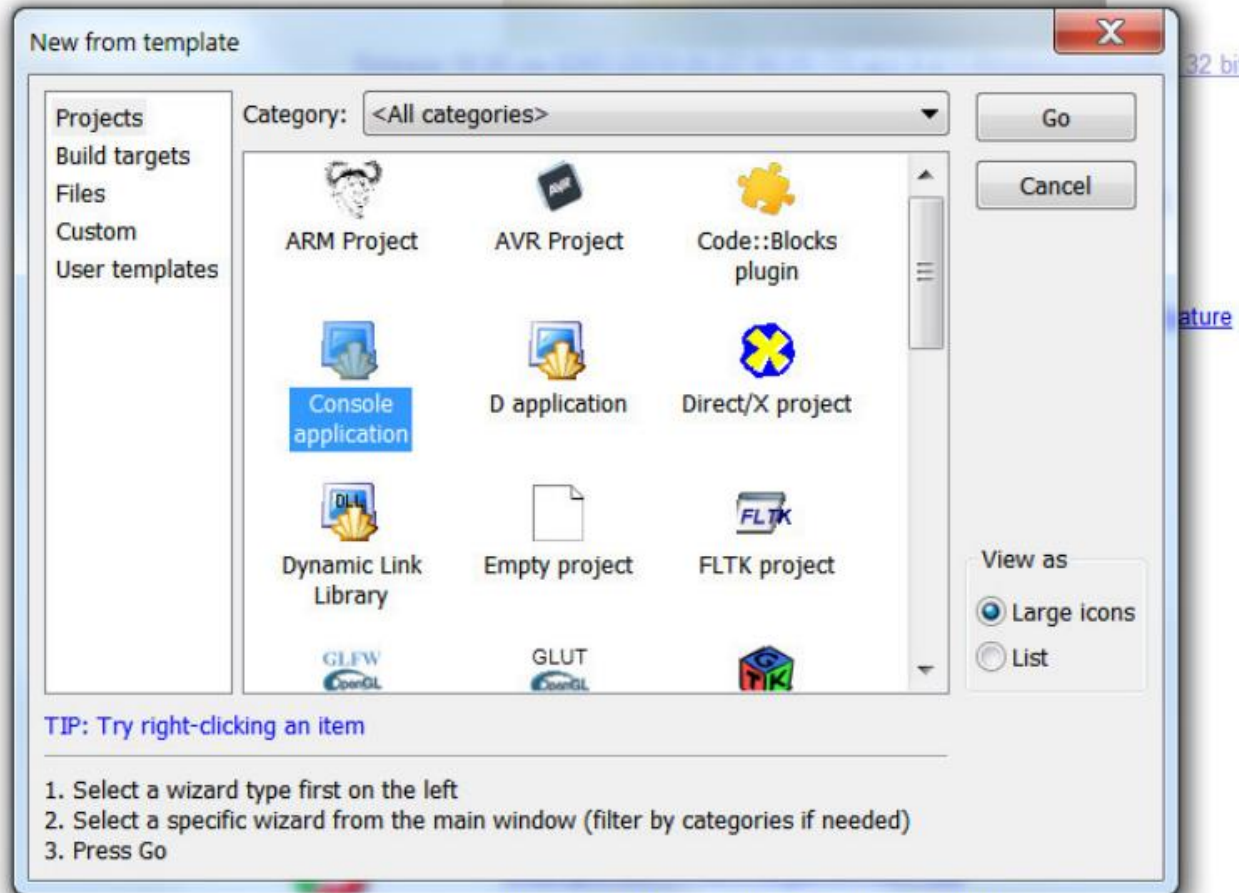




Code::Blocks

The open source, cross-platform IDE

<http://www.codeblocks.org>



[C:\Users\Ed\Dropbox\001 - Estrutura de dados -
ed\programas-c\estruturas\passando_structs_inteiras_para_funcoes.cpp](#)

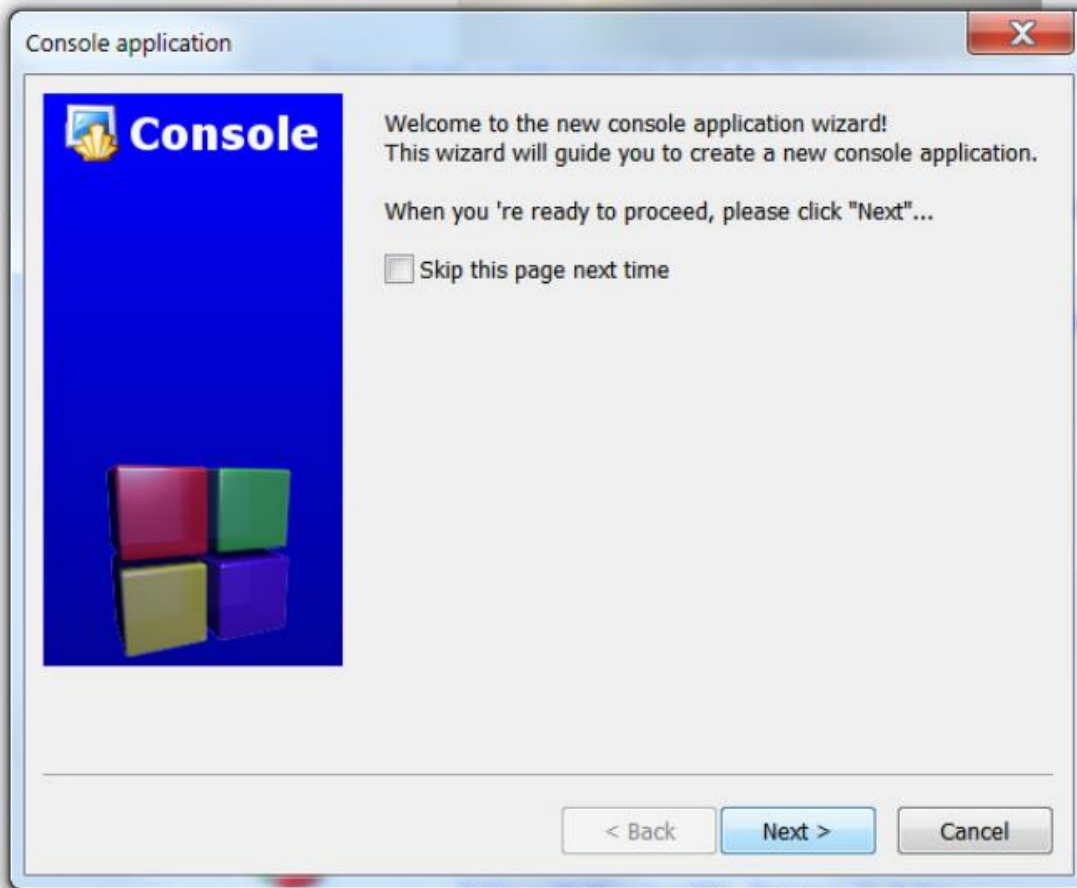
Escolha “Console Application”





Code::Blocks

The open source, cross-platform IDE.
<http://www.codeblocks.org>

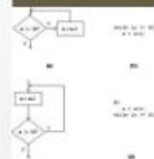


- 32 bit

feature

C:\Users\Ed\Dropbox\001 - Estrutura de dados -
ed\programas-c\estruturas\passando_structs_inteiros_para_funcoes.cpp

Clique em “next”

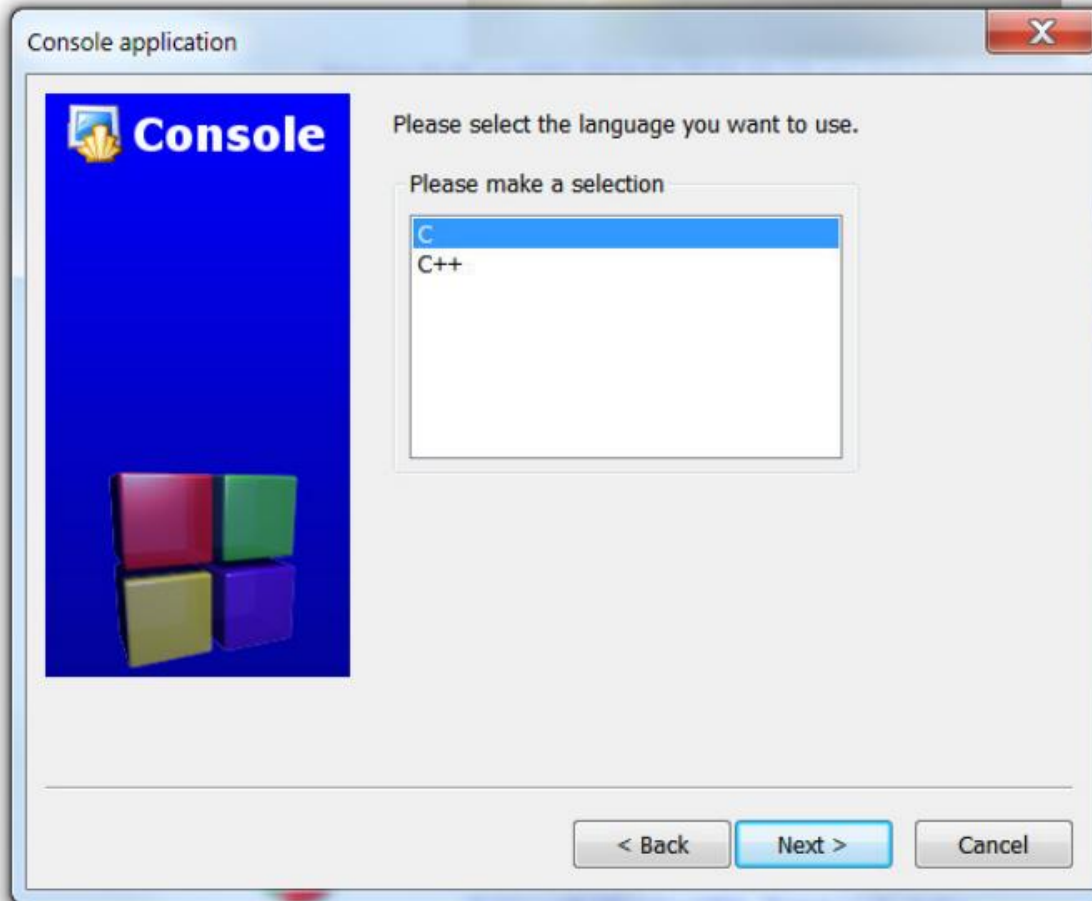




Code::Blocks

The open source, cross-platform IDE

<http://www.codeblocks.org>



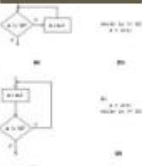
32 bit

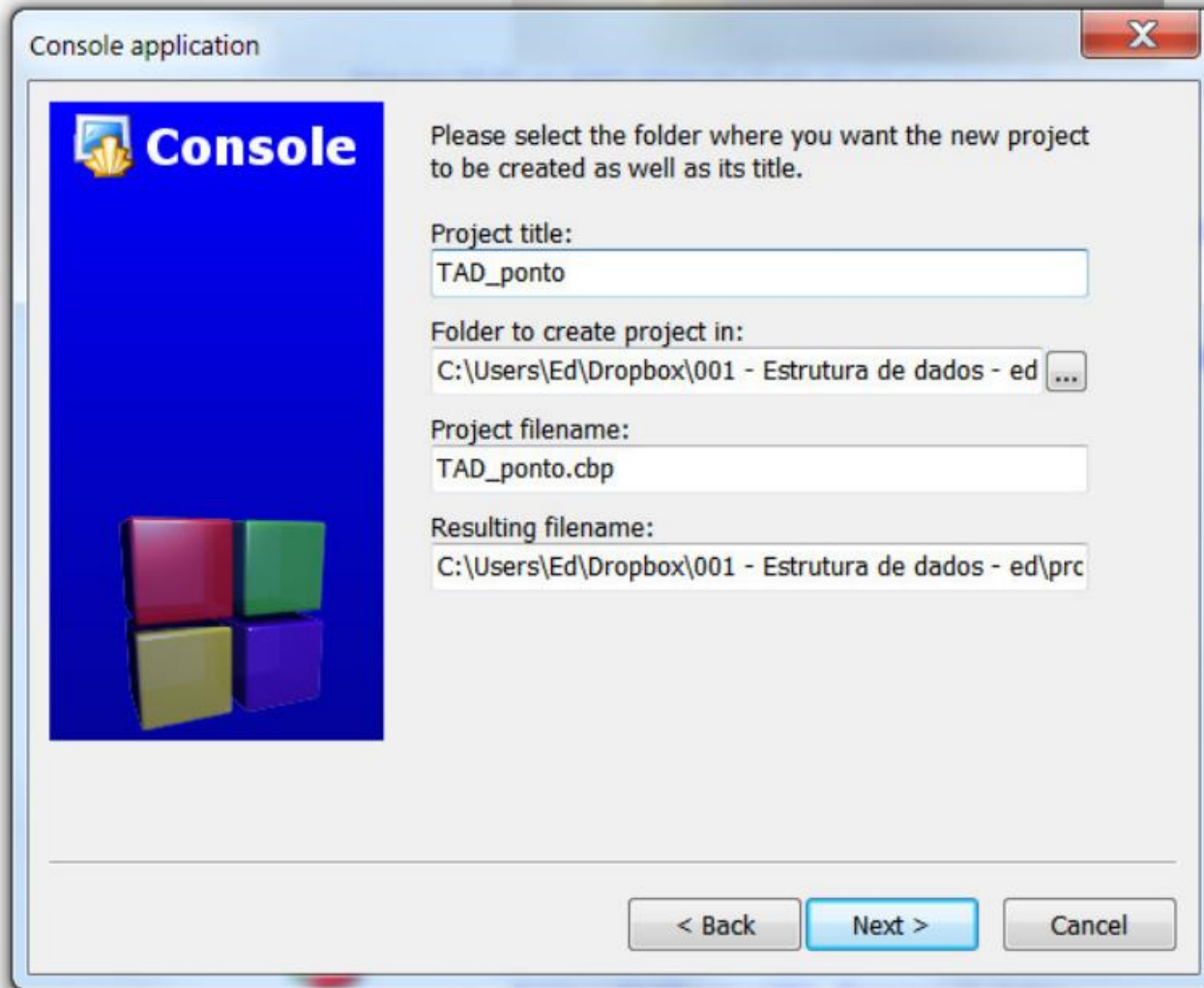
feature



[C:\Users\Ea\Dropbox\001 - Estrutura de dados - ed\programas-c\estruturas\passando_structs_inteiros_para_funcoes.cpp](#)

Escolha "C" (se salvou .c ou C++ se salvou .cpp)

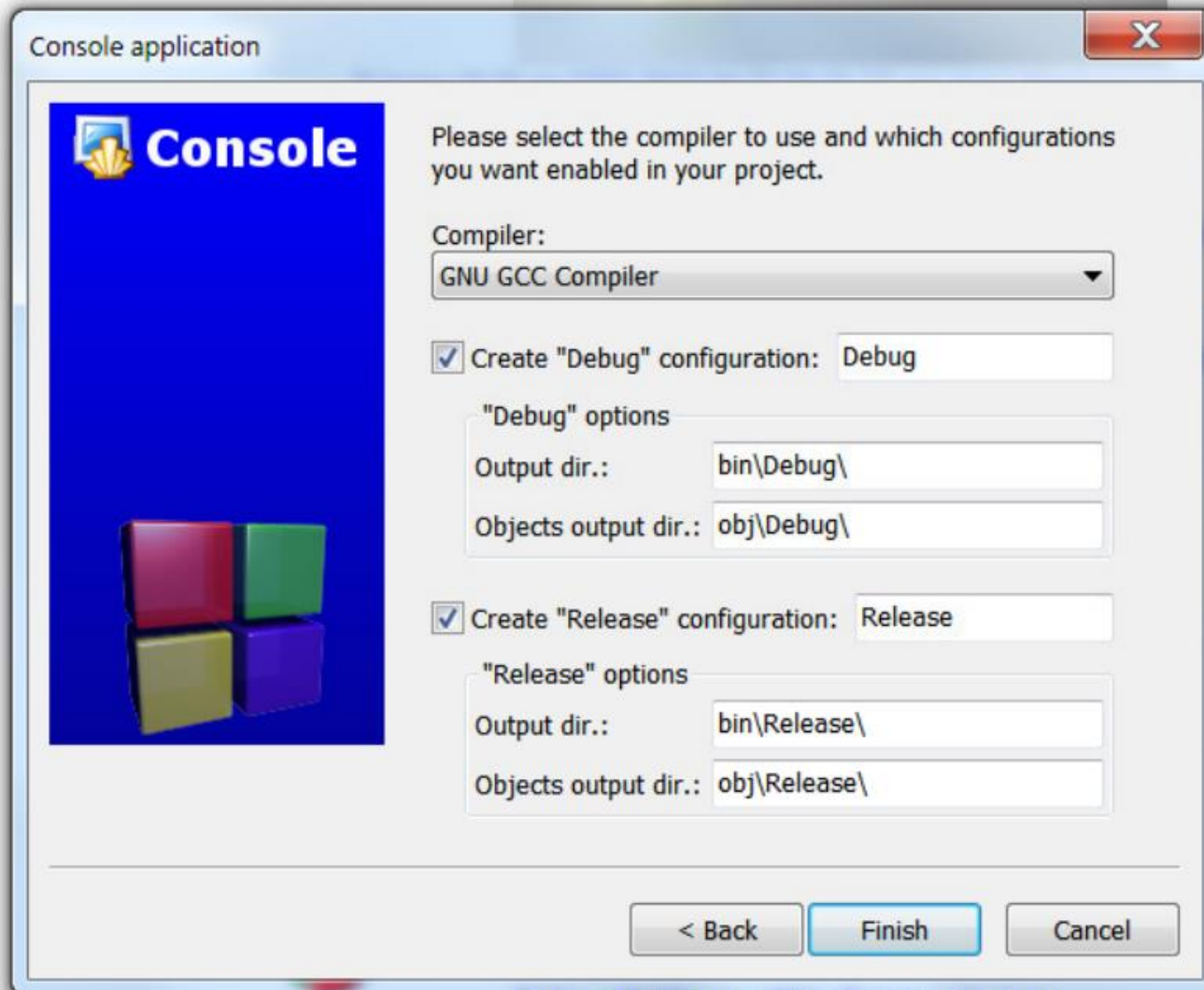




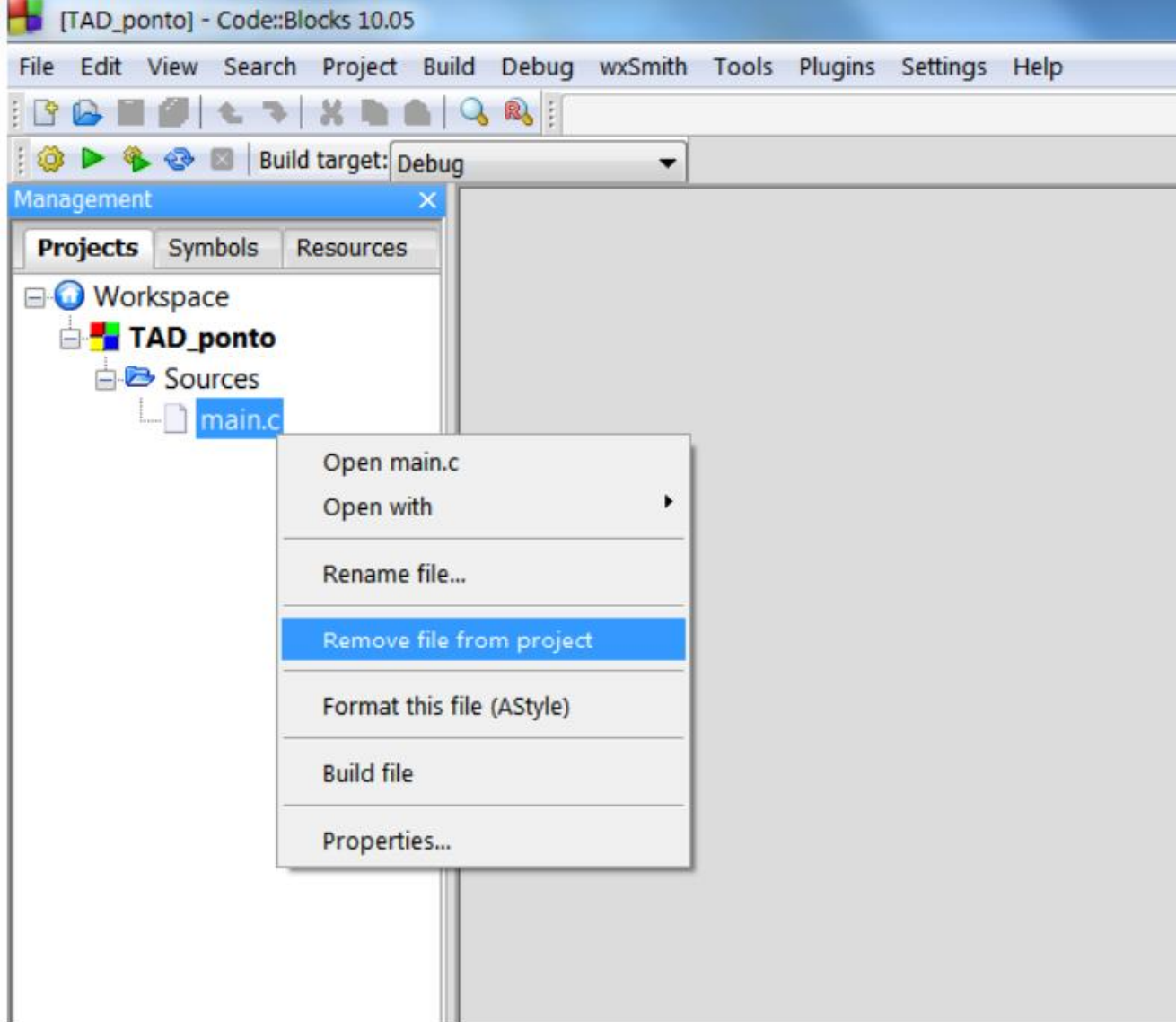
C:\Users\Ed\Dropbox\001 - Estrutura de dados -
ed\programas-c\estruturas\passando structs inteiras para funcoes

**Preencha com os dados adequados
(apenas o “project title”)**

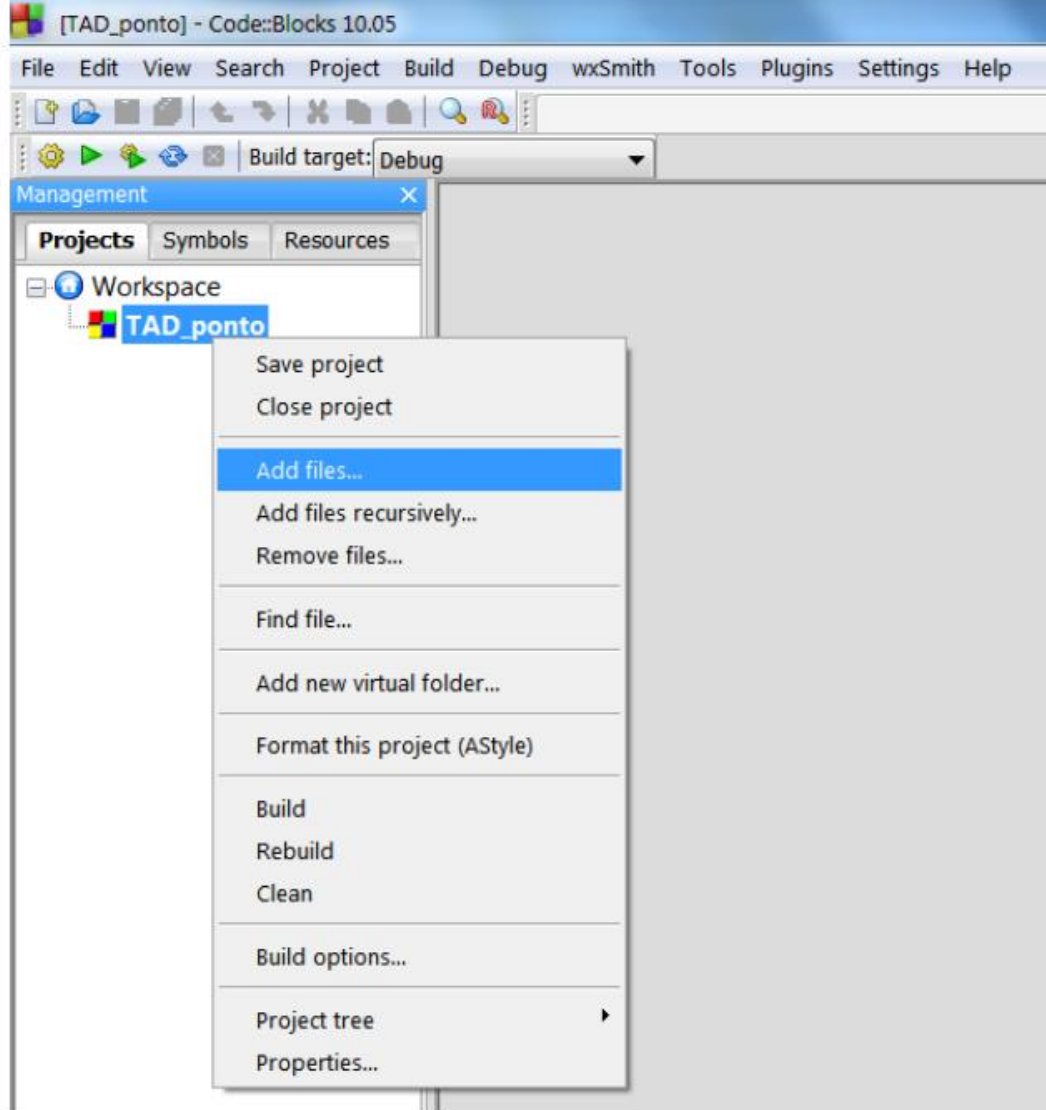




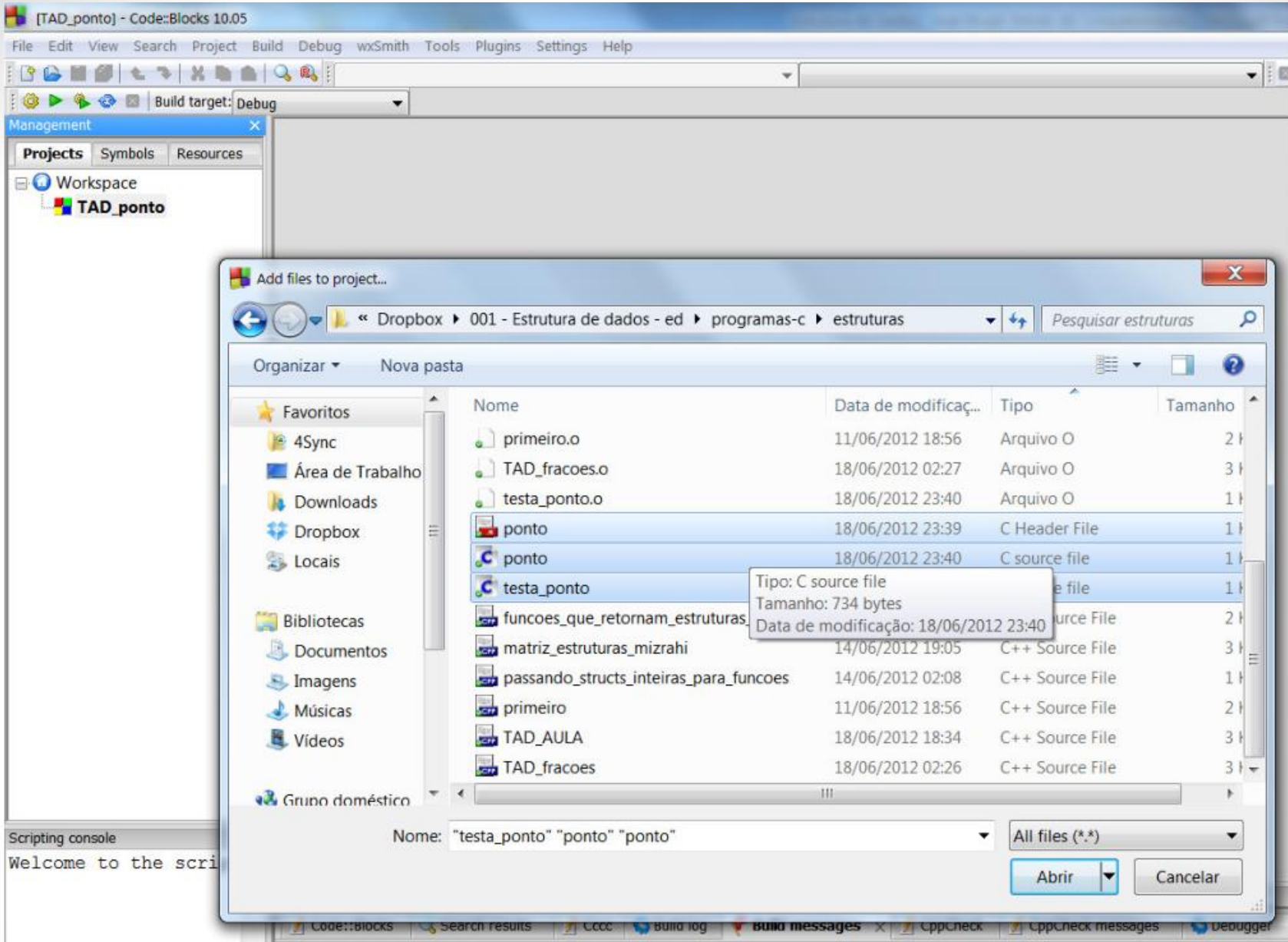
Escolha seu compilador instalado (não é necessário alterar outras opções)



Remova o arquivo criado pelo programa automaticamente (clique com o botão direito)

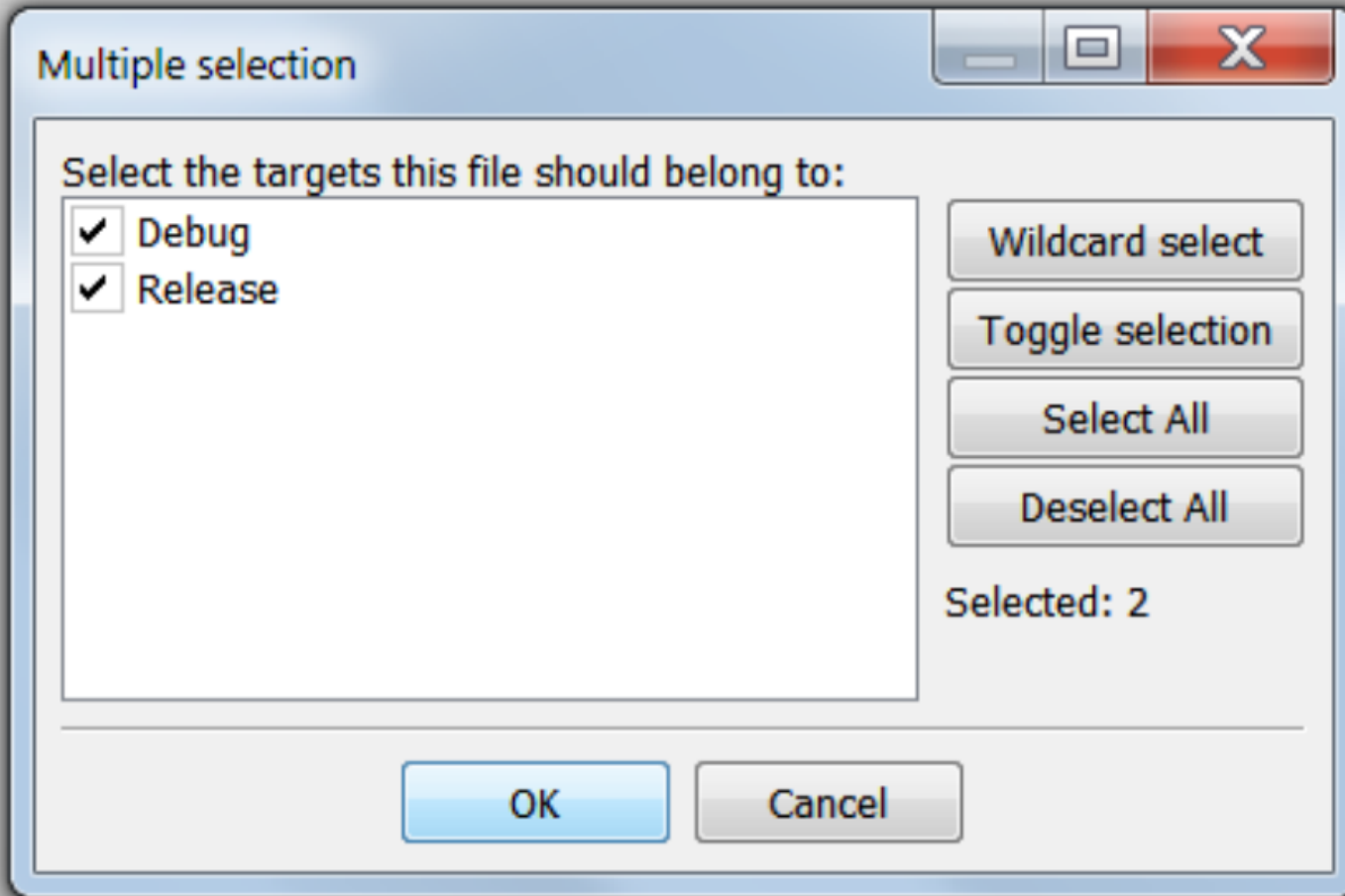


**Clique com o botão direito sobre seu projeto e
selecione “Add Files” (para adicionar seus arquivos
– ponto.c, ponto.h e testa_ponto.c)**

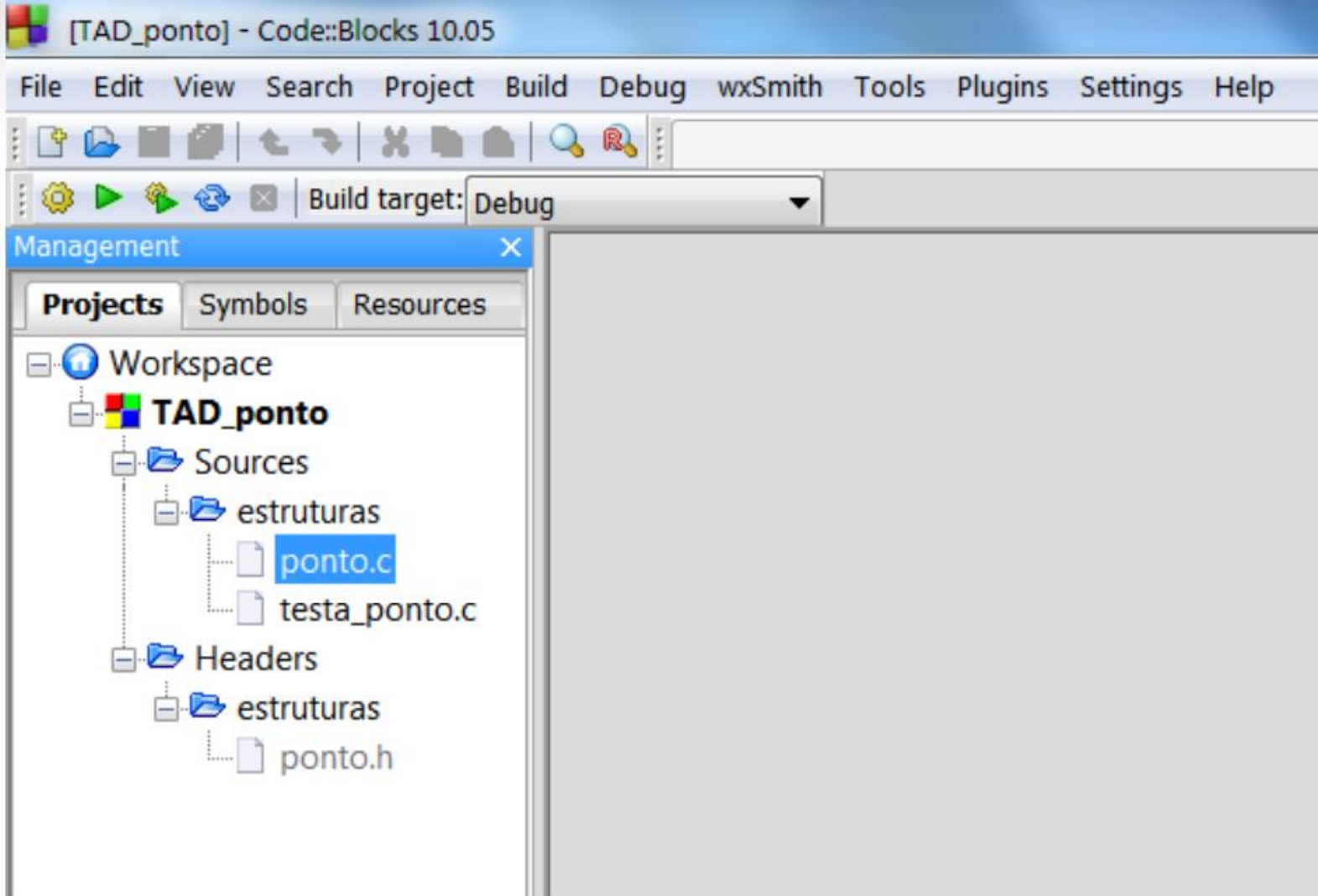


Localize e adicione seus arquivos

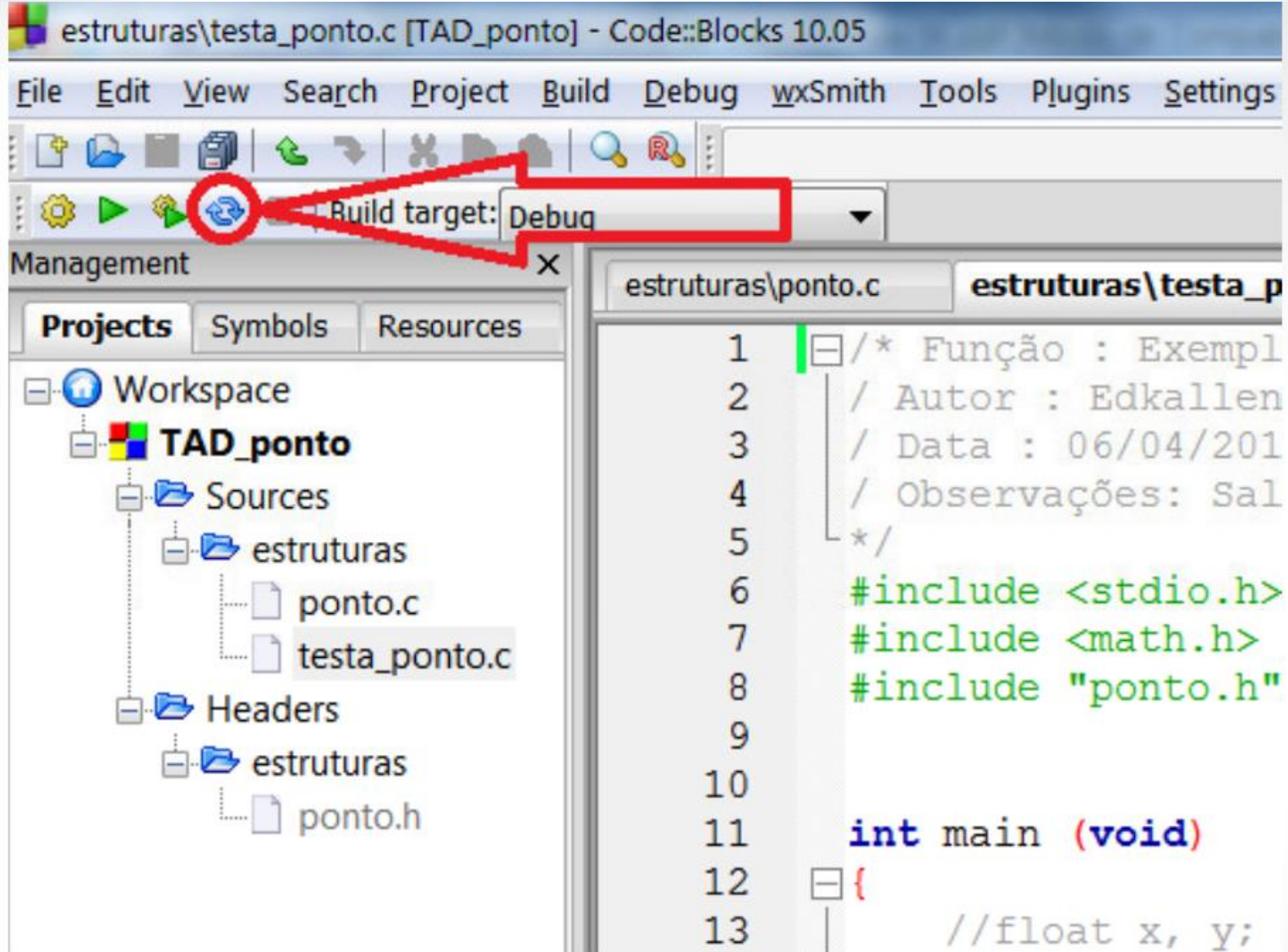




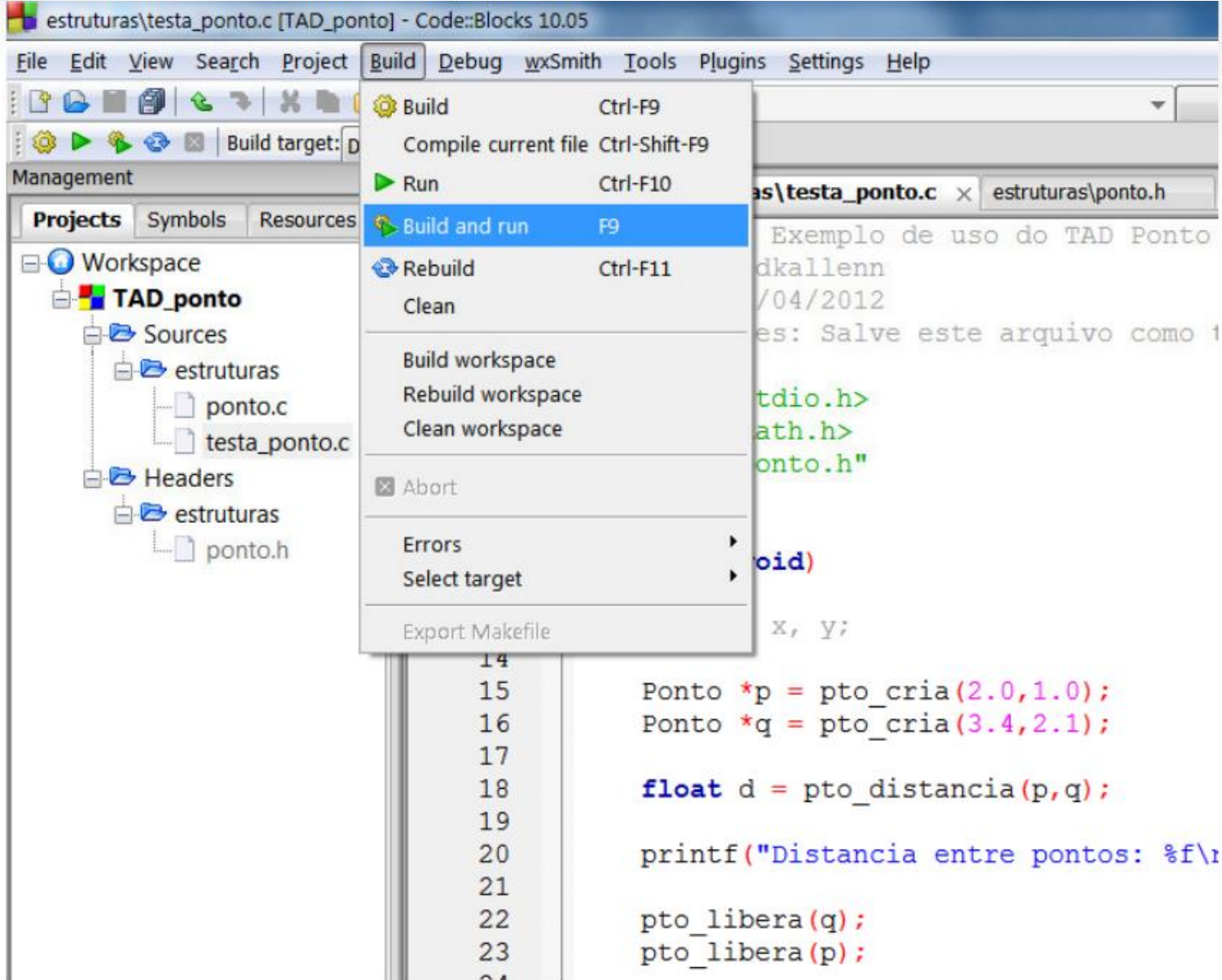
Clique “OK” na janela que aparece



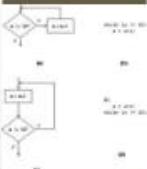
**Seus arquivos serão adicionados
ao projeto**



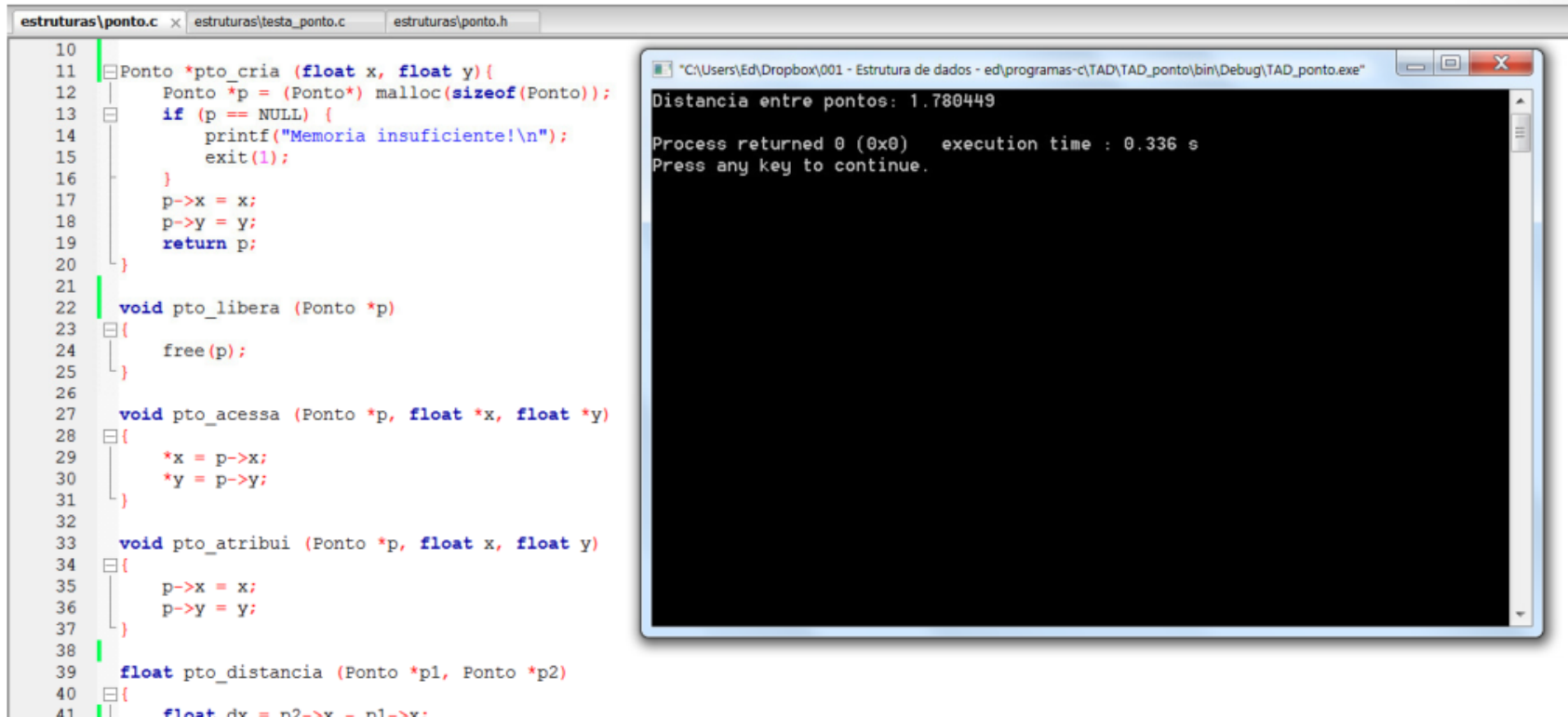
Abra-os (clique duas vezes sobre todos)
Clique em Rebuild. Depois, confirme.



Depois clique em "Build" → "Build and Run"



Eis o resultado:



The image shows a C program in a text editor and its execution in a command prompt window.

Code (estruturas\ponto.c):

```

10
11 Ponto *pto_cria (float x, float y) {
12     Ponto *p = (Ponto*) malloc(sizeof(Ponto));
13     if (p == NULL) {
14         printf("Memoria insuficiente!\n");
15         exit(1);
16     }
17     p->x = x;
18     p->y = y;
19     return p;
20 }
21
22 void pto_libera (Ponto *p)
23 {
24     free(p);
25 }
26
27 void pto_acessa (Ponto *p, float *x, float *y)
28 {
29     *x = p->x;
30     *y = p->y;
31 }
32
33 void pto_atribui (Ponto *p, float x, float y)
34 {
35     p->x = x;
36     p->y = y;
37 }
38
39 float pto_distancia (Ponto *p1, Ponto *p2)
40 {
41     float dx = p2->x - p1->x;

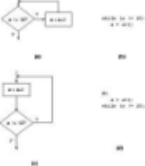
```

Execution Output (TAD_ponto.exe):

```

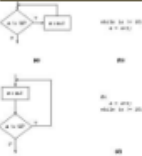
Distancia entre pontos: 1.780449
Process returned 0 (0x0)   execution time : 0.336 s
Press any key to continue.

```

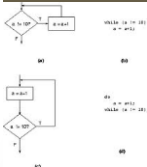


RESUMO

- **Módulo** → arquivo com funções que representam apenas parte da implementação de um programa completo
- **Arquivo objeto** → resultado de compilar um módulo geralmente com extensão .o ou .obj
- **Interface de módulo** → arquivo contendo apenas os protótipos das funções oferecidas pelo módulo, e os tipos de dados exportados pelo módulo
- **TAD** → define um novo tipo de dado e o conjunto de operações para manipular dados do tipo



**VER A LISTA DE
EXERCÍCIOS QUE ESTARÁ
DISPONÍVEL NO BLOG E NO
DROPBOX.**



- 