# Técnicas de Programação em C

## Representações Binárias

* %o
* Representa valor Octal
* %x ou %X
* Representa valor Hexadecimal integer
* %d ou %i
* Representa valor Decimal signed integer

**Exemplo 1:**

|  |
| --- |
| **#include <stdio.h>**  **int main()**  **{**  **int numero = 20;**  **printf("%o + %x = %04d", numero, numero, numero \* 5);**  **printf("\n");**  **printf("%o + %X = %04i", numero, numero, numero \* 5);**  **}**  **/\***  **Leia-me**  **%o --> Octal integer --> 20 em Octal = 24**  **%x ou %X --> Hexadecimal integer --> 20 em Hexadecimal = 14**  **%d ou %i --> Decimal signed integer --> 04d gera numero decimal com 4 casas --> multiplicado por 5 = 0100**  **\*/** |

* Operadores

Imagine o seguinte programa:

|  |
| --- |
| #include <stdio.h>  int main()  {  int i = 9;    i^=15;  printf("Para i^ seu valor: %d", i);  printf("\n");  **//imprime 6**  i>>=1;  printf("Para i>> seu valor: %d", i);  printf("\n");  **//imprime 3**  i<<=2;  printf("Para i<< seu valor: %d", i);  **//imprime 12**  printf("\n");    return 0;  } |

* ^ = é o operador XoR. *Ou exclusivo*

Resolução do programa acima para o operador “^”:

i=9;

i^=15; OU i = i^15;

9 em binário -> **1001**

15 em binário -> **1111**

**15-9 = 6 em binário -> 0110**

**Portanto, i=6**

* >> = operador de deslocamento de bits para a direita

Resolução do programa acima para o operador “>>”:

i=6;

i>>=1; OU i=i>>1

6 em binário -> 0**11**0

**Tem 1 deslocamento para a direita -> 0011**

**0011 é 3 em binário**

**Portanto, i=3**

* **<< =** operador de deslocamento de bits para a esquerda

Resolução do programa acima para o operador “>>”:

i=3;

i<<=2; OU i=i<<2

3 em binário -> **0011**

**Tem 2 deslocamento para a esquerda-> 0011**

**1100 é 12 em binário**

**Portanto, i=12**

Imagine o seguinte programa:

|  |
| --- |
| #include <stdio.h>  int main()  {  int i=~9;  printf("%d",i);  **//imprime -10**  printf("\n");  return 0; } |

* ~ = operador complemento

Resolução do programa acima para o operador “~”:

i=9;

9 em binário -> 0000 0000 0000 1001

**~ (inverte os bits) -> 1111 1111 1111 0110**

**OBS: (é um numero negativo porque inicia com 1) – Lembrando: 1 (-) | 0 (+)**

**Somente para o entendimento:**

C utiliza complemento 2 para representar números negativos, ou seja, primeiro vamos inverter os bits e segundo somar por 1. Então:

0000 0000 0000 1001

**+ 1**

**0000 0000 0000 1010**

**É gerado o valor 10**

**Assim, inverte os bits**

**1111 1111 1111 0110**

**Portanto, i=-10**

Imagine o seguinte programa:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main()  {  int i = pow(2,16) - 1;    if(!i)  printf("Verdadeiro %d", i);  **//imprime “Verdadeiro 65535”**  else  printf("Falso %d", i);    return 0;  } |

* i = é o operador que representa o valor decimal

Na primeira linha de código é uma função exponencial, sendo dois elevado a dezesseis menos um que vai ser igual a 65535.

No caso “**i”** representa o valor binário 1111 1111 1111 1111.

Mas como existe o operador “!” que é o NOT, o valor binário vai ser contrário, no caso, 0000 0000 0000 0000.

Portanto, na condicional vai perguntar se esse valor não é zero, e se não for zero ele entra no IF, se não for zero entra o ELSE. Como o valor é 65535 e não é zero, logo vai entrar no IF gerando valor verdadeiro.

Imagine o seguinte programa:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main()  {  int bit1 = 1;  int bit2 = 8;    int resultado = bit1 ^ bit2;    printf("Resultado: %d", resultado, "\n");  **//imprime 9**  return 0;  } |

* ^ = é o operador XoR. *Ou exclusivo*

*Temos bit 1 = 0000 0000 0000 0001*

*Temos bit 9 = 0000 0000 0000 1000*

*Portanto, vai ser somados os bits, imprimindo o valor 9 que em bits é*

*0000 0000 0000 1001.*

# Palavras Chaves Padrão

A linguagem C possui um total de 32 palavras conforme definido pelo padrão ANSI, que são elas:

|  |  |  |  |
| --- | --- | --- | --- |
| * **auto** * **break** * **case** * **char** * **const** * **continue** * **default** * **do** | * **double** * **else** * **enum** * **extern** * **float** * **for** * **goto** * **if** | * **int** * **long** * **register** * **return** * **short** * **signed** * **sizeof** * **static** | * **struct** * **switch** * **typedef** * **union** * **unsigned** * **void** * **volatile** * **while** |

É importante lembrar que todas as palavras reservadas são escritas em minúsculo e não podem ser utilizadas para outro propósito. Alguns compiladores incluem outras palavras reservadas como, **asm**, **cdecl**, **far**, **fortran**, **huge**, **interrupt**, **near**, **pascal**, **typeof**.

## Especificador *auto*

O especificador de classe de armazenamento ***auto*** define variáveis locais. Raramente são usados, pois todas as variáveis do C são auto por definição.

## Especificador *extern*

O ***extern*** define variáveis que serão usadas em um arquivo apesar de terem sido declaradas em outro.

Quando o sistema é separado em vários programas, pode-se ter o problema de acesso a certas variáveis globais, pois a definição da mesma pode estar em um programa fonte e é necessário acessar estas variáveis em outro programa fonte.

Como na Linguagem C deve-se sempre definir uma variável antes de usá-la, quando ocorrer á situação acima descrita deve ser indicado no programa que irá usar a variável que a mesma está definida em outro programa.

Para se fazer isto basta colocar a palavra **extern** na frente da definição da variável juntamente com a definição do seu tipo e nome. Feito isto está sendo indicado para o compilador o necessário para que não sejam gerados erros, e indicado que a variável já foi definida em outro arquivo fonte.

|  |
| --- |
| Programa Principal extern |
| #include <stdio.h>  **/\* Declaração do protótipo da função. Mesmo que a função não esteja no código fonte,é importante "informar" ao compilador que esta função existe, senão ocorrerá erro na compilação \*/**  void imprime\_soma (void);  **/\* Declaração das variáveis públicas, ou seja, elas estarão disponíveis para uso em todo programa \*/**  int iValor\_a;  int iValor\_b;    int main (void)  {    printf ("Entre com os valores:");  scanf ("%d %d", &iValor\_a, &iValor\_b);  imprime\_soma();    return 0;  } |
| Programa extern auxiliar |
| #include <stdio.h>  **/\* Declarando que EXISTEM, em outro programa, as variáveis públicas, ou seja, elas estarão disponíveis para uso em todo o programa. \*/**  extern int iValor\_a;  extern int iValor\_b;    void imprime\_soma(void)  {  **/\* uso das variáveis públicas \*/**  printf ("Soma %d\n", iValor\_a + iValor\_b); return;  } |

## Especificador *static* para casos de variáveis globais

Como padrão toda variável definida dentro de uma função é alocada na pilha interna de execução da função. Ao final da função a pilha é liberada, liberando assim a memória alocada pela variável. Na próxima chamada à função é feita uma nova alocação na pilha assim por diante.

Deve-se indicar através da palavra **static** na definição de uma variável, quando for necessário que a variável local de uma função permaneça com o seu valor mantido, permitindo assim que na próxima chamada utilizar o valor anterior.

* O funcionamento das variáveis declaradas como static depende se estas são globais ou locais.
* Variáveis globais static funcionam como variáveis globais dentro de um módulo, ou seja, são variáveis globais que não são (e nem podem ser) conhecidas em outros módulos.
* Isto é útil se quisermos isolar pedaços de um programa para evitar mudanças acidentais em variáveis globais.

Digamos que tenhamos dois programas usando o especificador extern.

programaPrincipal.c programaExterno.c

extern int count;

função(){

Estrutura da função

}

int **static** count;

main{

Estrutura do programa

}

No exemplo acima o programa “programaExterno.c” não vai encontrar a variável ***count***, uma vez que ela vai estar definida como static, logo deixa de existir como uma variável externa global e sim como local .

Para resolver esse problema poderia criar mais um programa e nela definir uma funcão para chamar a variável global ***count***. Veja a imagem abaixo:

programaPrincipal.c programaSecundario.c programaExterno.c

int count;

função(){

Estrutura da função

}

int extern count;

função(){

Estrutura da função

}

int **static** count;

main{

Estrutura do programa

}

## Especificador *static* para casos de variável local

Variáveis locais ***static*** são variáveis cujo valor é mantido de uma chamada da função para a outra. Veja o exemplo:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  **//Usando o especificador static**  int comStatic(void){    static int num = 0;  num++;  return num;  }  **//Não Usando o especificador static**  int semStatic(void){    int num = 0;  num++;  return num;  }  int main()  {  **//Imprimindo com Static o valor num=3**  int a;    a = comStatic();  a = comStatic();  a = comStatic();    printf("Resultado com Static: %d",a);    printf("\n\n");    **//Imprimindo sem Static o valor num=1**  int b;    b = semStatic();  b = semStatic();  b - semStatic();    printf("Resultado sem Static: %d",b);    } |

**Com static**

Como ***num*** é uma variável static, o seu valor vai continuar entre as chamadas da função, ignorando o valor inicial definido como zero (num=0) logo vai continuar a fazer as três chamadas da função, ou seja, fazendo o chinês:

**Primeira chamada da função -> num=0**

**num ++**

**num =1**

**Segunda chamada da função -> num = 1**

**num++**

**num = 2**

**Terceira chamada da função -> num = 2**

**num++**

**num = 3**

**Sem static**

Na variável sem static o num quando voltar para função vai retornar com o valor definida inicial, ou seja, não vai ignorar o valor inicial da variável num (num=0), fazendo o chinês:

**Primeira chamada da função -> num=0**

**num ++**

**num =1**

**Segunda chamada da função -> num = 0**

**num++**

**num = 1**

**Terceira chamada da função -> num = 0**

**num++**

**num = 1**

## Especificador *register*

O computador tem a memória principal e os registradores da CPU. As variáveis (assim como o programa como um todo) são armazenadas na memória. O modificador ***register*** diz ao compilador que a variável em questão deve ser, **se possível**, usada em um registrador da CPU.

* Garante que aumente a Velocidade
* A variável não pode ser global
* Pedido do programador, mas não significa que vai ser feito.

programaPrincipal.c

main{

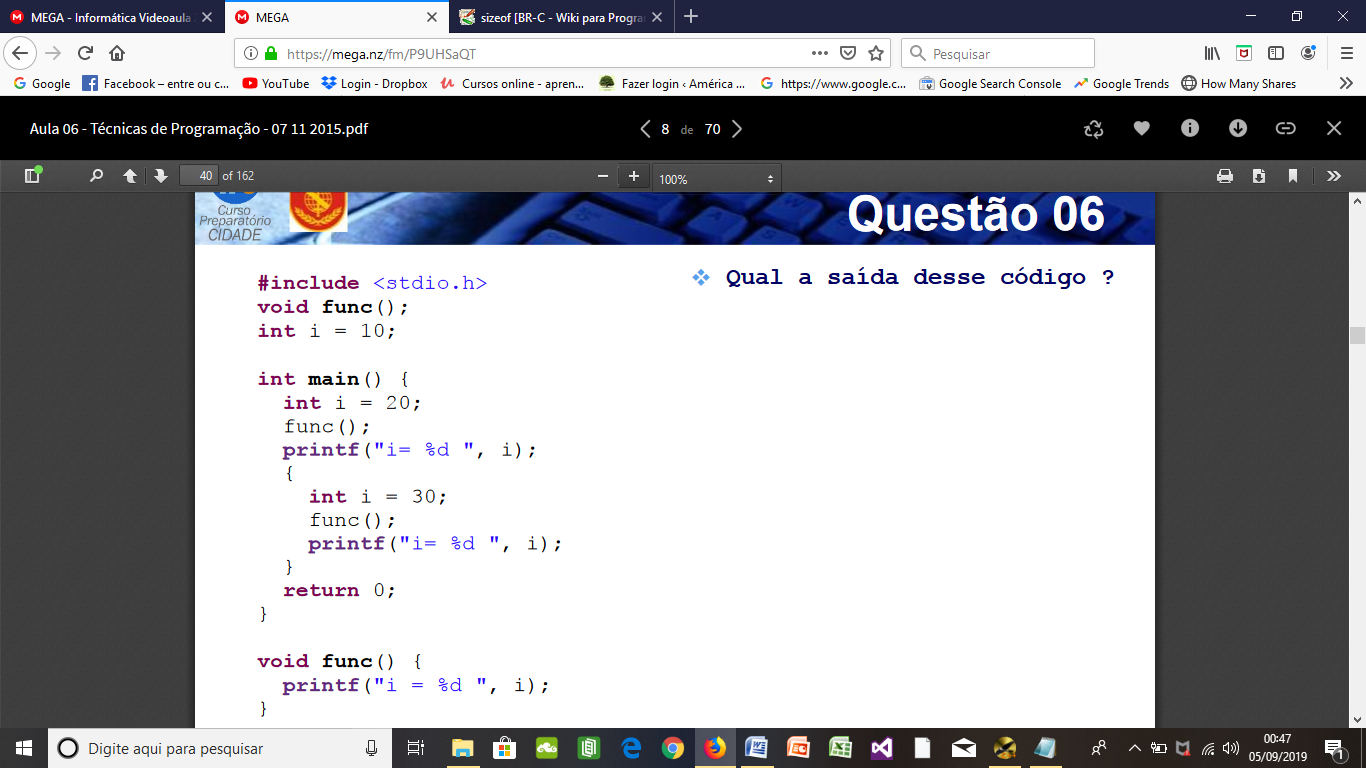
int register nome\_variavel;

Estrutura do programa

}

# Entendendo a saída do código

Veja o seguinte código:



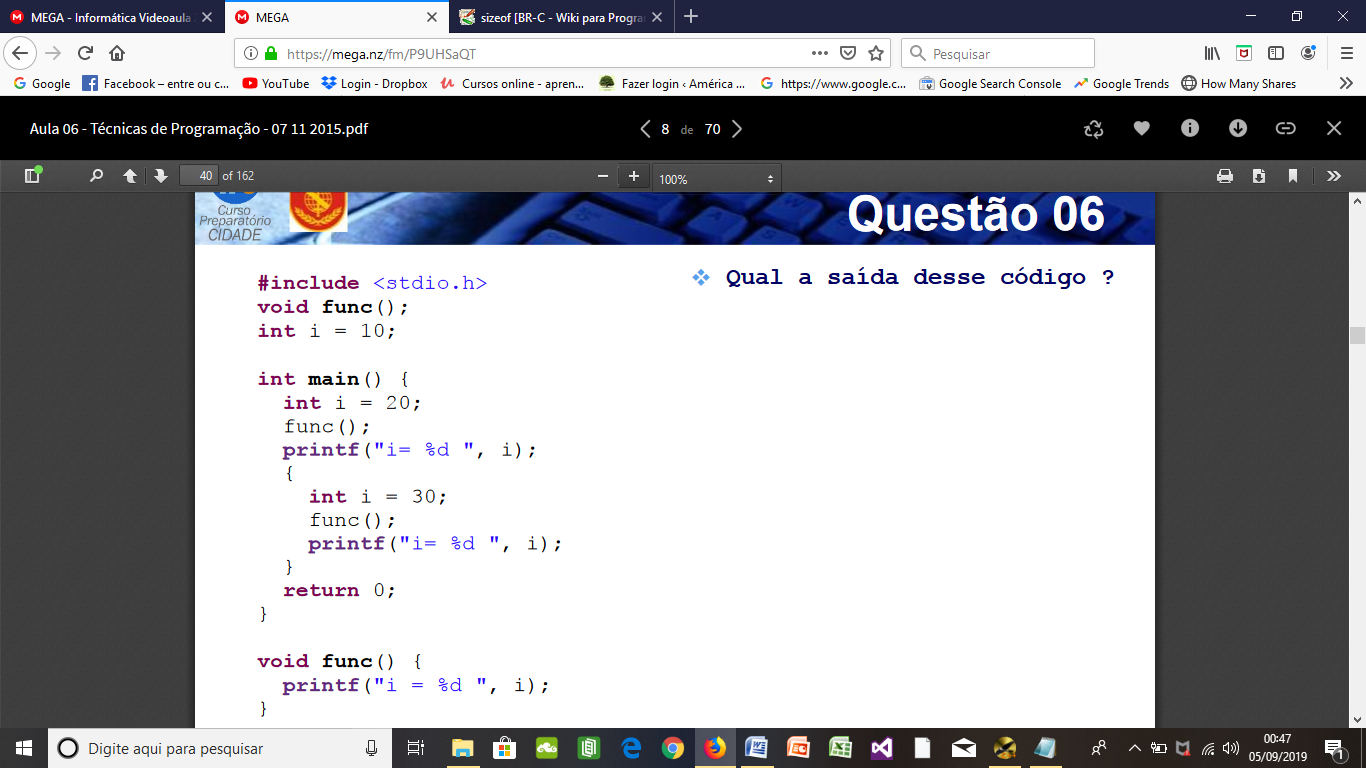
Dados importantes:

1º Verificar os escopos, se são global ou local:

Temos três escopos globais, são elas:

* **int i = 10;**
* **int main(){...}**
* **void func(){...}**
* **OBS: Tudo que está dentro da função main é local. Repare que a variável int i dentro do main é uma variável local (int i =10) e a variável int i fora do main é uma variável global (int i = 20).**

**Fazendo o chinês:**

****

func{

i = 10;

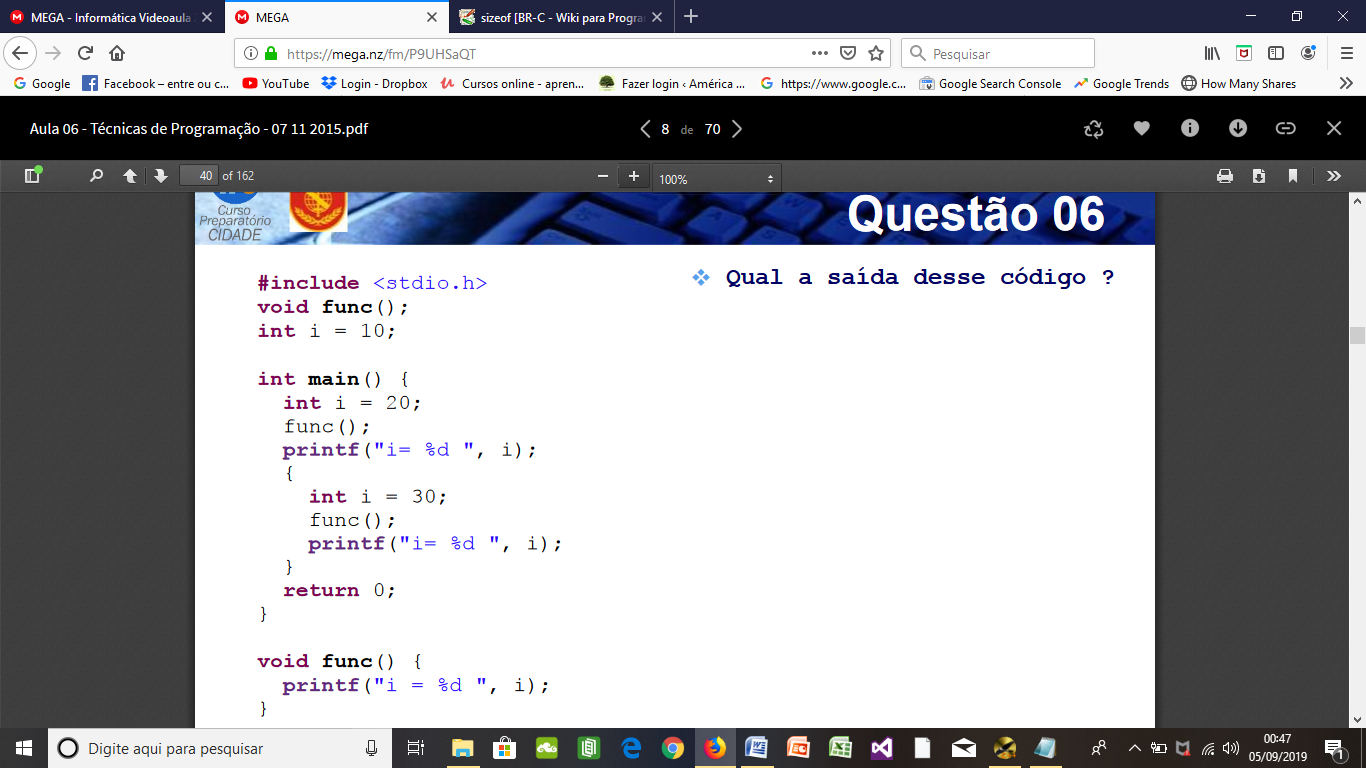
**printf (imprime 10)**

}

}

int i = 20;

func();

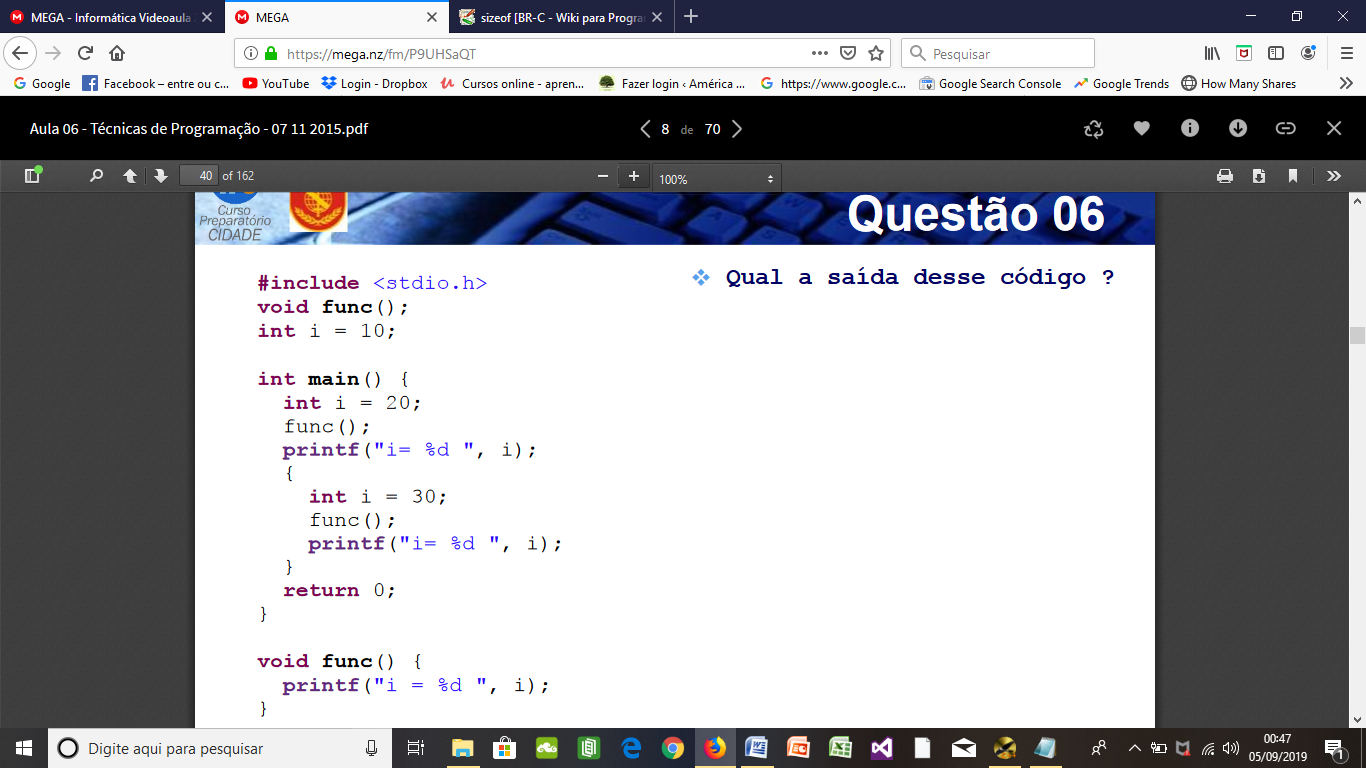
****

int i = 20;

func();



**printf(imprime 20)**

****

int i = 20;

func();

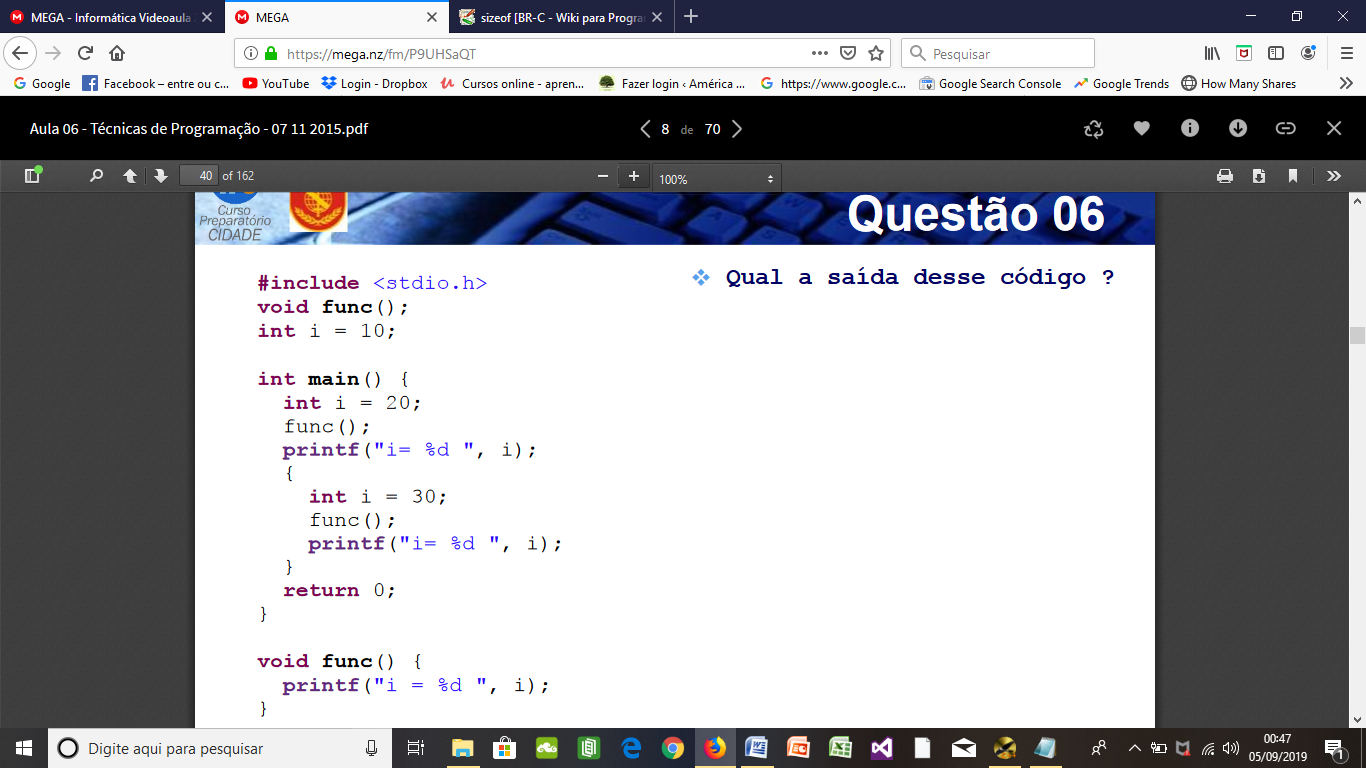


printf(imprime 20)

int = 30;

func();



****

int i = 20;

func();



printf(imprime 20)

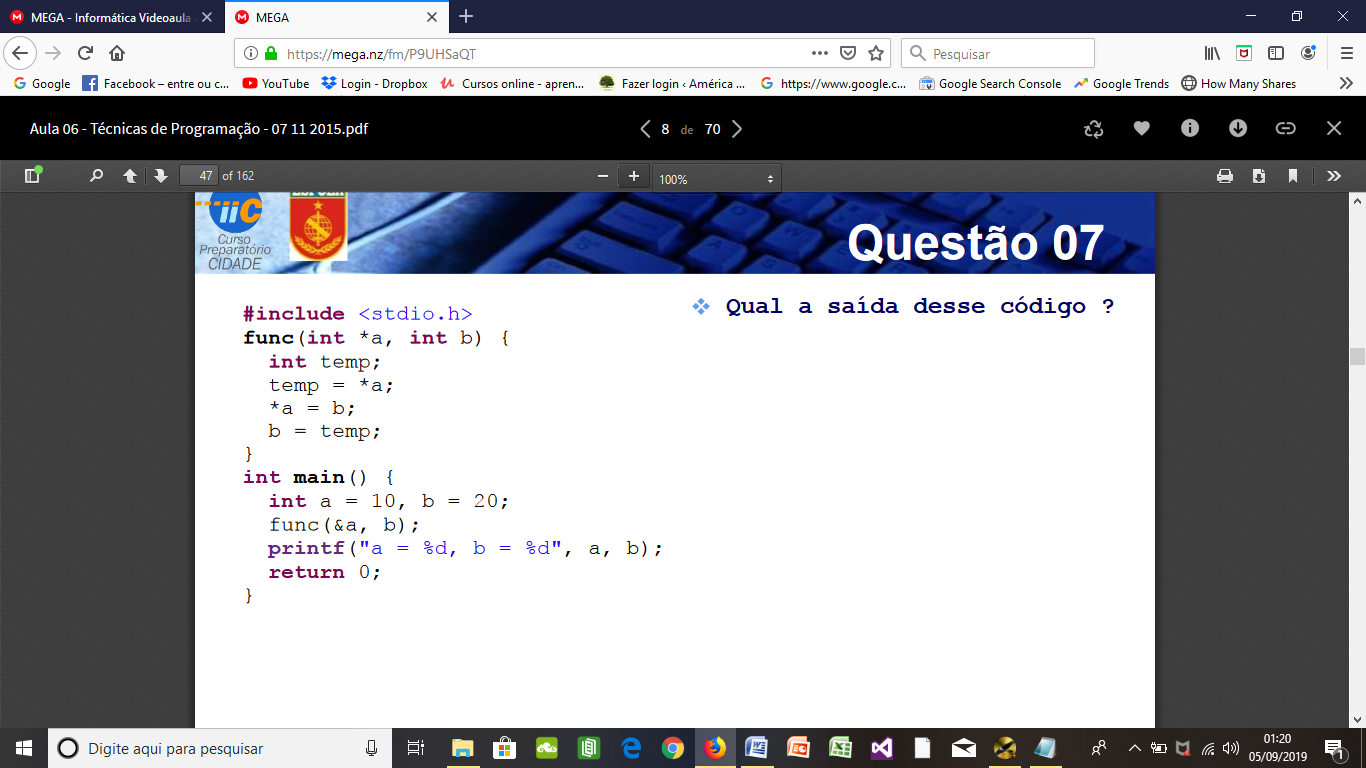
int = 30;

func();



**printf(imprime (30)**

**Exemplo 2:**

****

**Fazendo o chinês:**

**Dentro do main{**

**a =10, b = 20;**

**func (&a, b);**

func(int \*a, int b){

temp = 10;

\*a = 20;

b = 10;

**printf(a = 20, b = 20);}**

**Portanto, imprime a=10 e b=20.**

**OBS:** Repare no *printf* que os valores a serem impressos somente o “**a**” existe ponteiro de referencia, ou seja, a variável local cujo valor “**a**” que está dentro do main é a mesma variável global que vai estar na função, portanto, se na função o valor “**a**” sofre alguma alteração, vai ser considerado o mesmo a ser impresso dentro do main.

Para referenciar um ponteiro é usado um asterístico (\*), no caso “***\*a”.***

***Lembrando: Se houver ponteiro de referencia na variável vai ser sempre o mesmo valor, não importa como vai ser manipulado o mesmo, podendo está dentro ou fora do main, ou seja, na função ou dentro do main o valor vai ser o mesmo.***