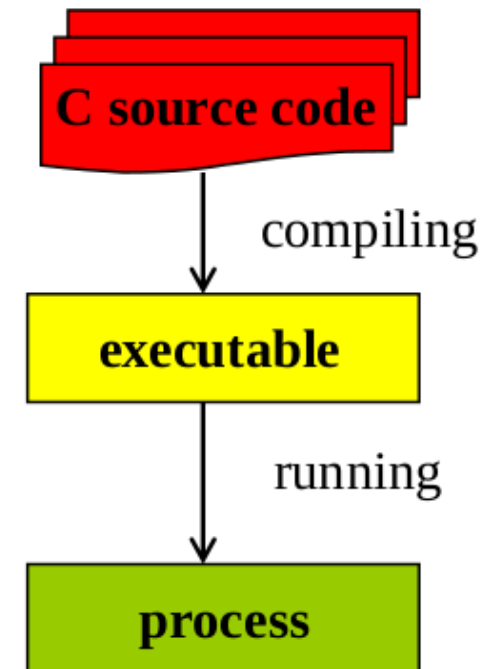




Arquitetura da Memória

Memória e Programas/Processos

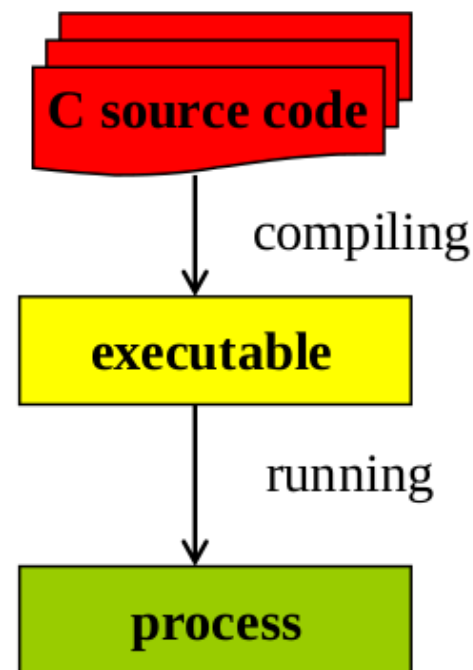
- **Memória:** armazena **padrões de bits** que podem ser **dados** ou **instruções**.
- Programas “rodam” na memória. O que significa isso?
 - **Código fonte:**
 - Declarações, algoritmos, etc.
 - Armazenados em arquivos (.c, .h)
 - **Executável:**
 - Binário gerado pelo compilador
 - Armazenado em arquivo (a.out,)



Memória e Programas/Processos

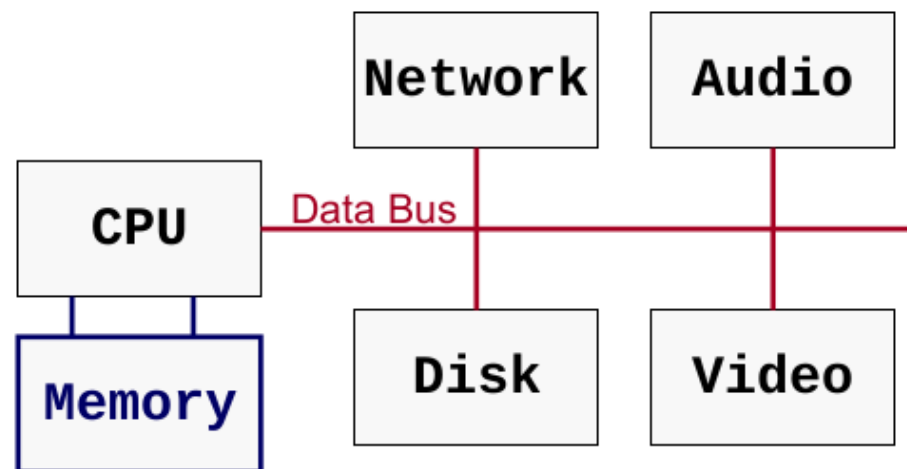
- Processo:

- Instância de um programa em execução (programa em execução)
 - Tem seu próprio espaço de endereços de memória
 - Tem seu próprio ID e estado de execução
- Gerenciado pelo SO



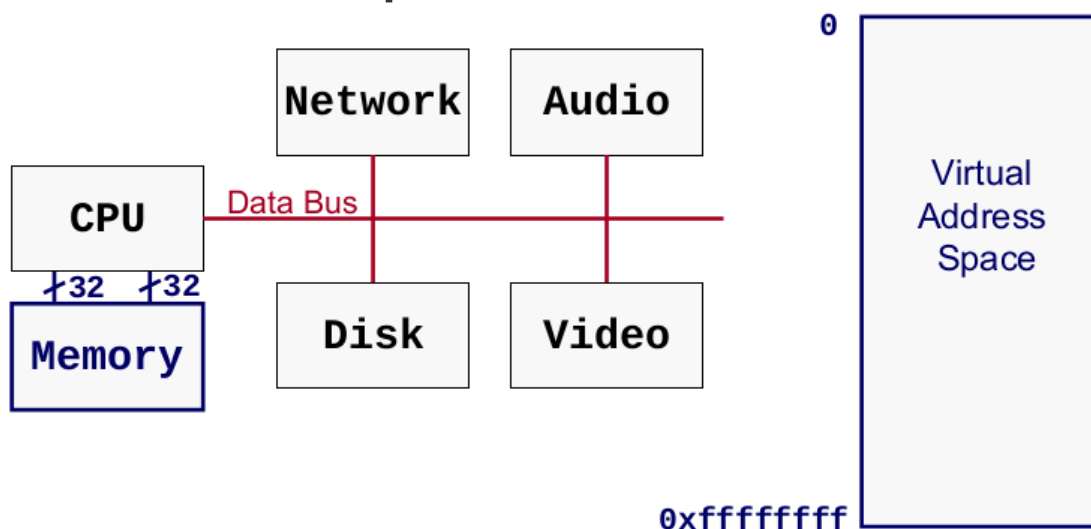
Memória Principal x Virtual

- **Memória principal:**
 - Área de armazenamento para variáveis, dados, instruções, etc.
 - Compartilhada entre diversos processos
 - Existe “de verdade”



Memória Principal x Virtual

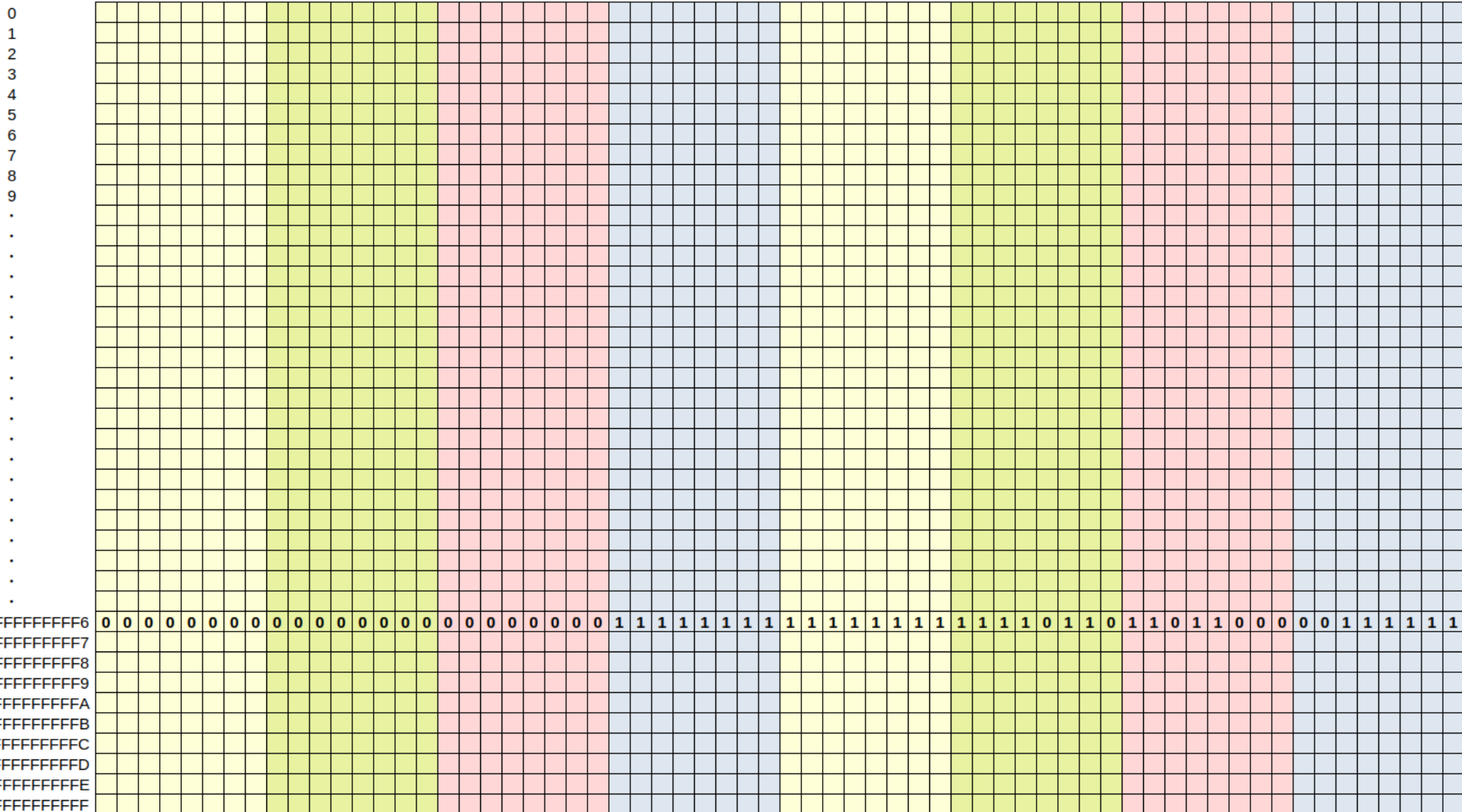
- **Memória virtual:**
 - É uma abstração dos recursos de armazenamento realmente disponíveis em um computador
 - “Ilusão” para os usuários: todo o **espaço de endereçamento virtual** está presente, independente da quantidade de RAM física real



Memória Principal x Virtual

- **Espaço de endereçamento virtual:**
 - 64 bits: endereços alocados por 8 bytes
 - 32 bits: endereços alocados por 4 bytes
 - 16 bits: endereços alocados por 2 bytes(menor unidade endereçável pela CPU: 1 byte (8 bits))
- **Word** (palavra): é uma quantidade fixa de bits que é manipulada pelo computador como uma unidade.
- O que significa isso??

Memória Principal x Virtual



Memória Principal x Virtual

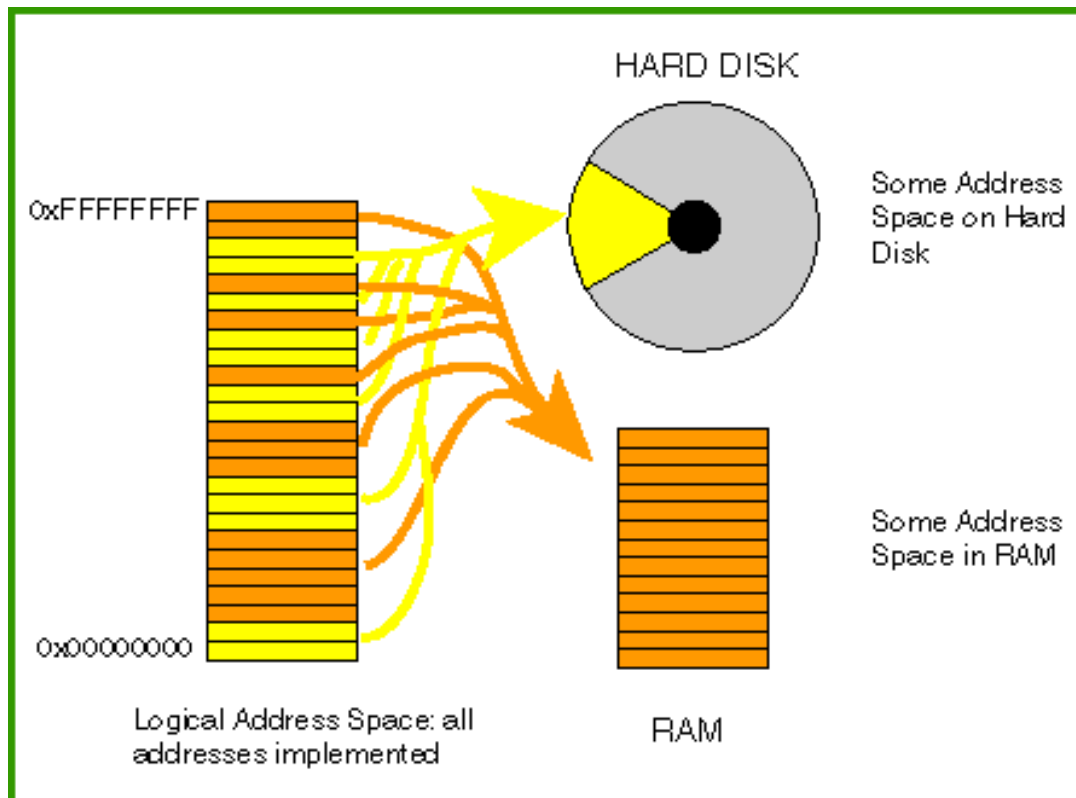
Em computadores com arquitetura de 64 bits:

- Cada Word tem 64 bits (8 bytes);
- Cada Word pode armazenar $2^{64} = 18446744073709551616$ números diferentes, variando entre 0 e $2^{64} - 1$ (0 a 18446744073709551615; 0 a FFFFFFFFFFFFFFFFFF em hexadecimal);
- Isso significa que computadores de 64 bits podem endereçar até 16 EiB de memória ($18446744073709551616 / 1024^6 = 16$).

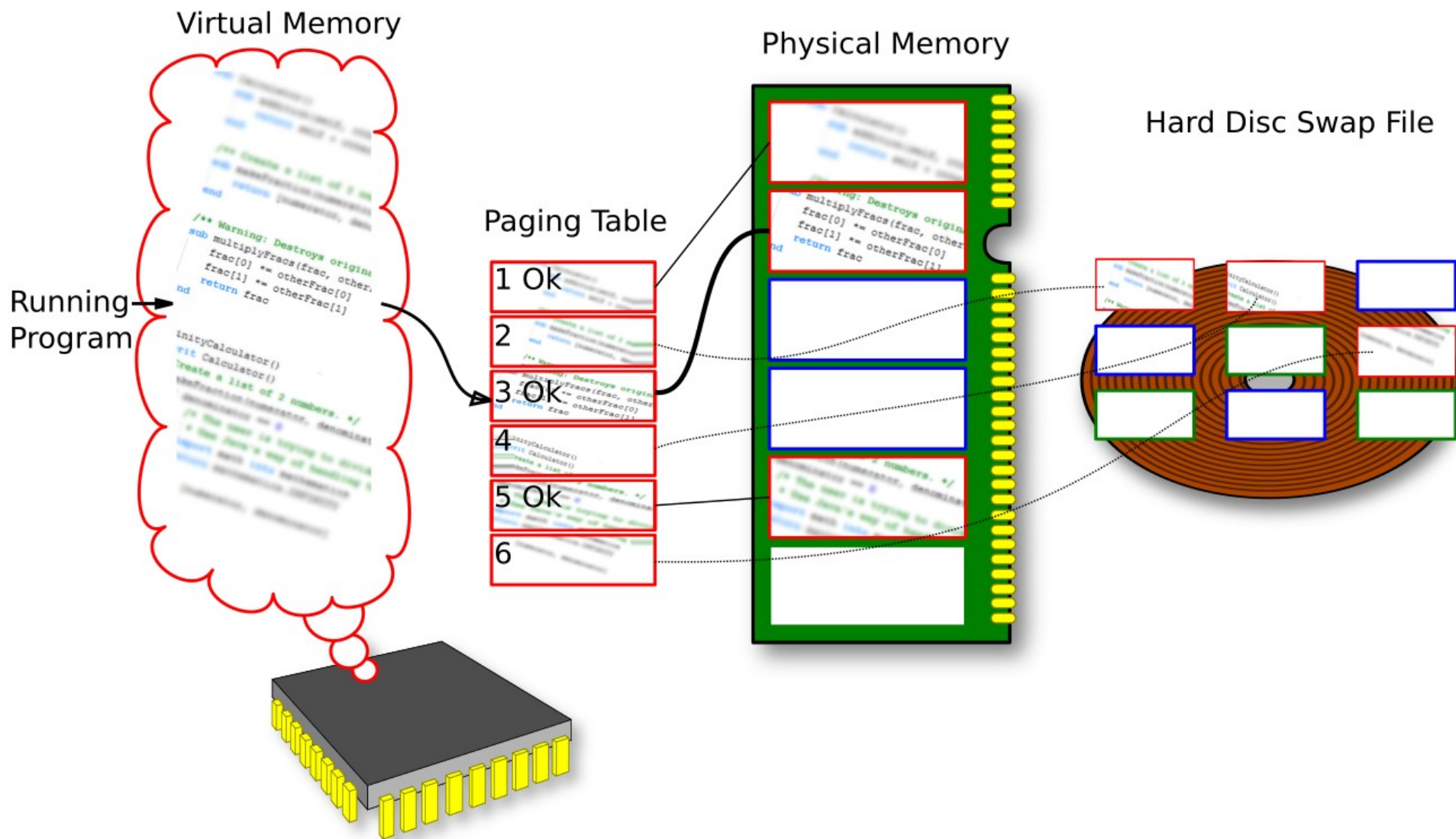
O endereço de memória FFFFFFFFFFFFFFFFFF6 (18446744073709551606) armazena um número binário que representa OUTRO ENDEREÇO de memória, o endereço FFFFF6D83F (1099511027775).

Memória Principal x Virtual

- Como pode isso?
 - TODO o espaço de endereçamento está presente, independente da quantidade de RAM física real.



Memória Principal x Virtual



Memória Principal x Virtual

- Aff...

Physical Memory Limits: Windows Vista

The following table specifies the limits on physical memory for Windows Vista.

Version	Limit on X86	Limit on X64
Windows Vista Ultimate	4 GB	128 GB
Windows Vista Enterprise	4 GB	128 GB
Windows Vista Business	4 GB	128 GB
Windows Vista Home Premium	4 GB	16 GB
Windows Vista Home Basic	4 GB	8 GB
Windows Vista Starter	1 GB	

Memória Principal x Virtual

- Aff...

Physical Memory Limits: Windows 7

The following table specifies the limits on physical memory for Windows 7.

Version	Limit on X86	Limit on X64
Windows 7 Ultimate	4 GB	192 GB
Windows 7 Enterprise	4 GB	192 GB
Windows 7 Professional	4 GB	192 GB
Windows 7 Home Premium	4 GB	16 GB
Windows 7 Home Basic	4 GB	8 GB
Windows 7 Starter	2 GB	N/A

Memória Principal x Virtual

- Aff...

Physical Memory Limits: Windows 10

The following table specifies the limits on physical memory for Windows 10.

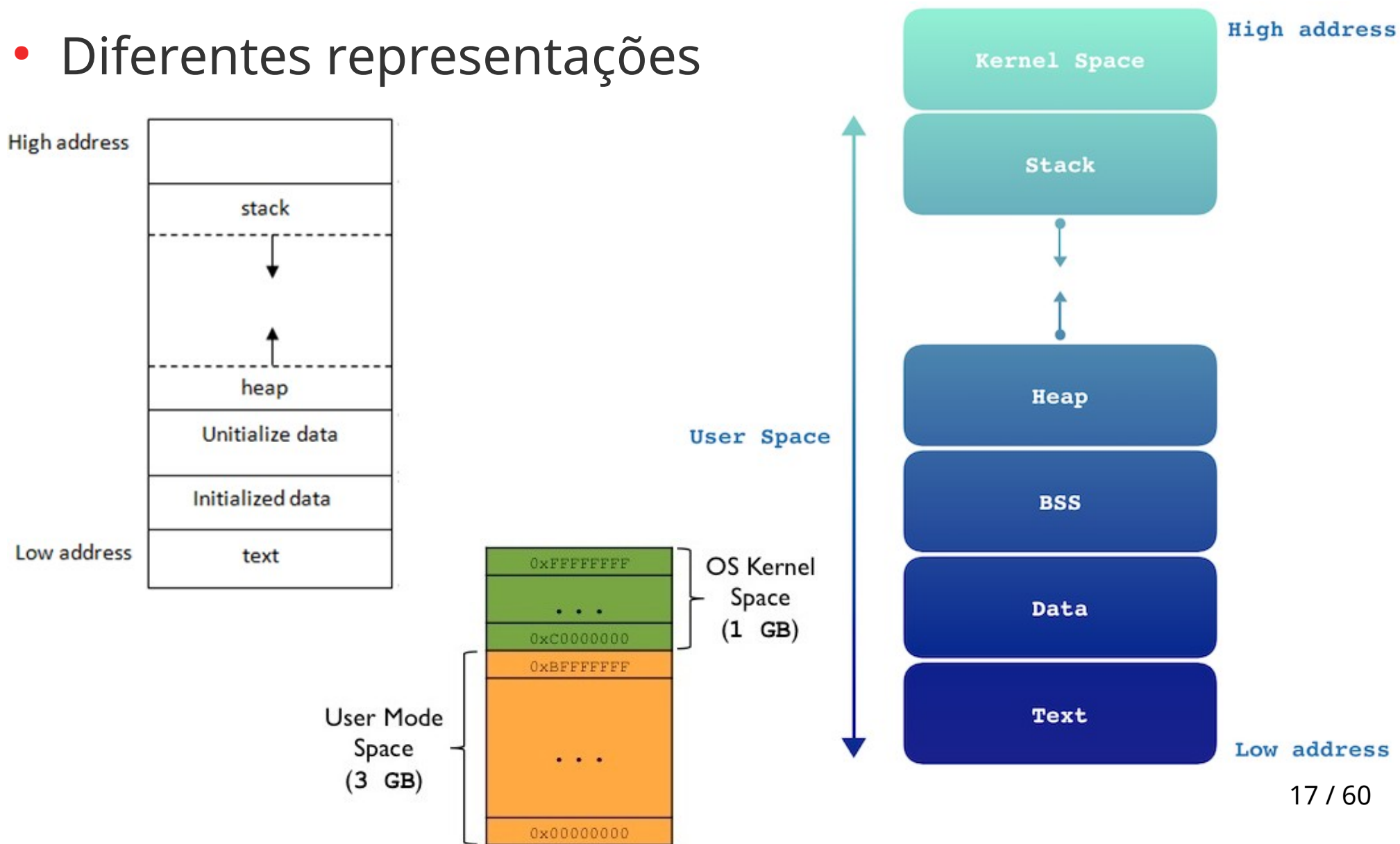
Version	Limit on X86	Limit on X64
Windows 10 Enterprise	4 GB	6 TB
Windows 10 Education	4 GB	2 TB
Windows 10 Pro for Workstations	4 GB	6 TB
Windows 10 Pro	4 GB	2 TB
Windows 10 Home	4 GB	128 GB

Segmentos da memória

- Quando um programa executa, o processamento é realizado em 2 grandes espaços da memória (que interagem entre si durante o processamento do programa):
 - **Kernel Space**: recursos reservados para o sistema; só pode ser acessada por processos de usuário através de chamadas de sistema.
 - **User Space**: recursos reservados para o usuário; o programa em execução consegue acessar diretamente. Dividido em diversos segmentos.

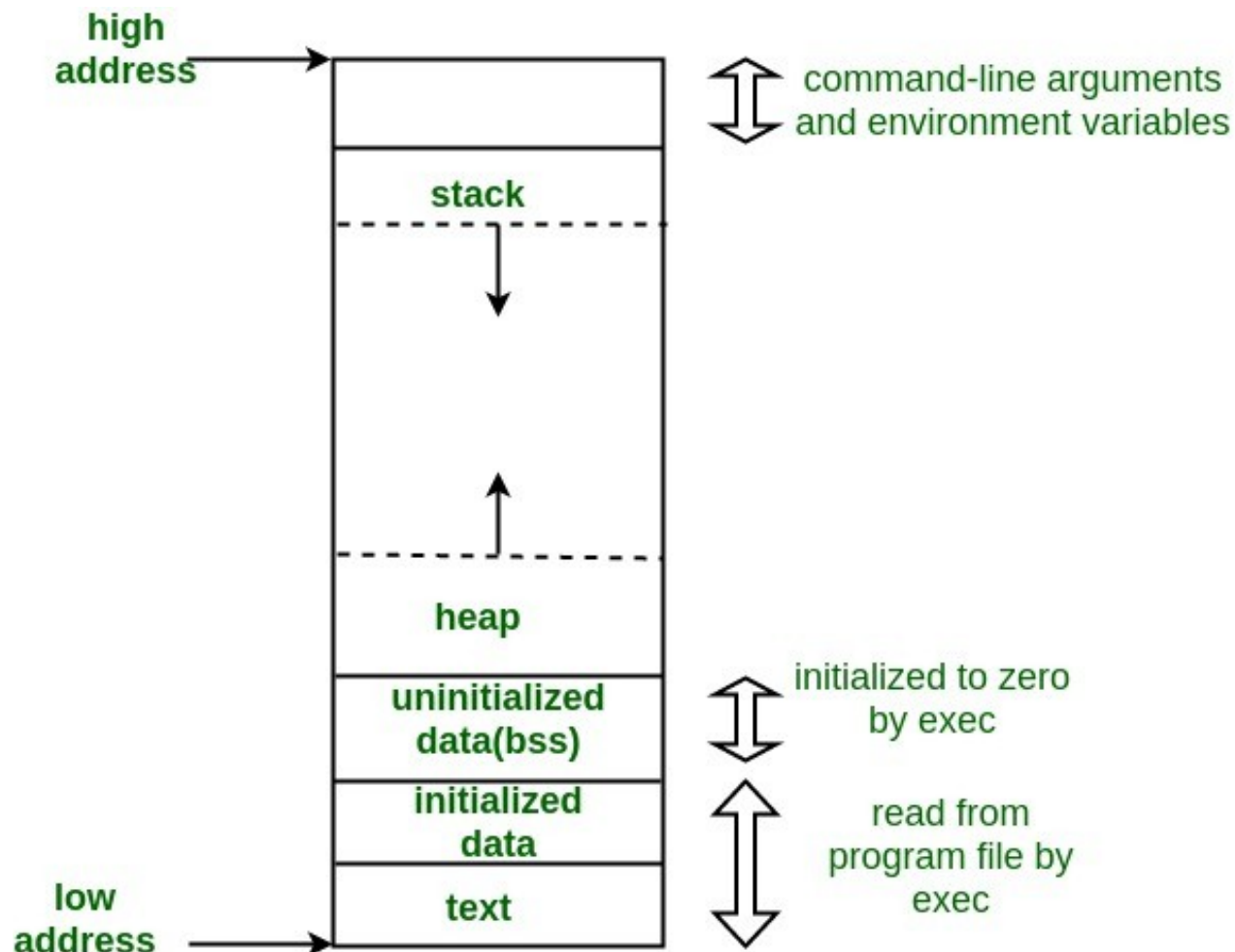
Segmentos da memória

- Diferentes representações



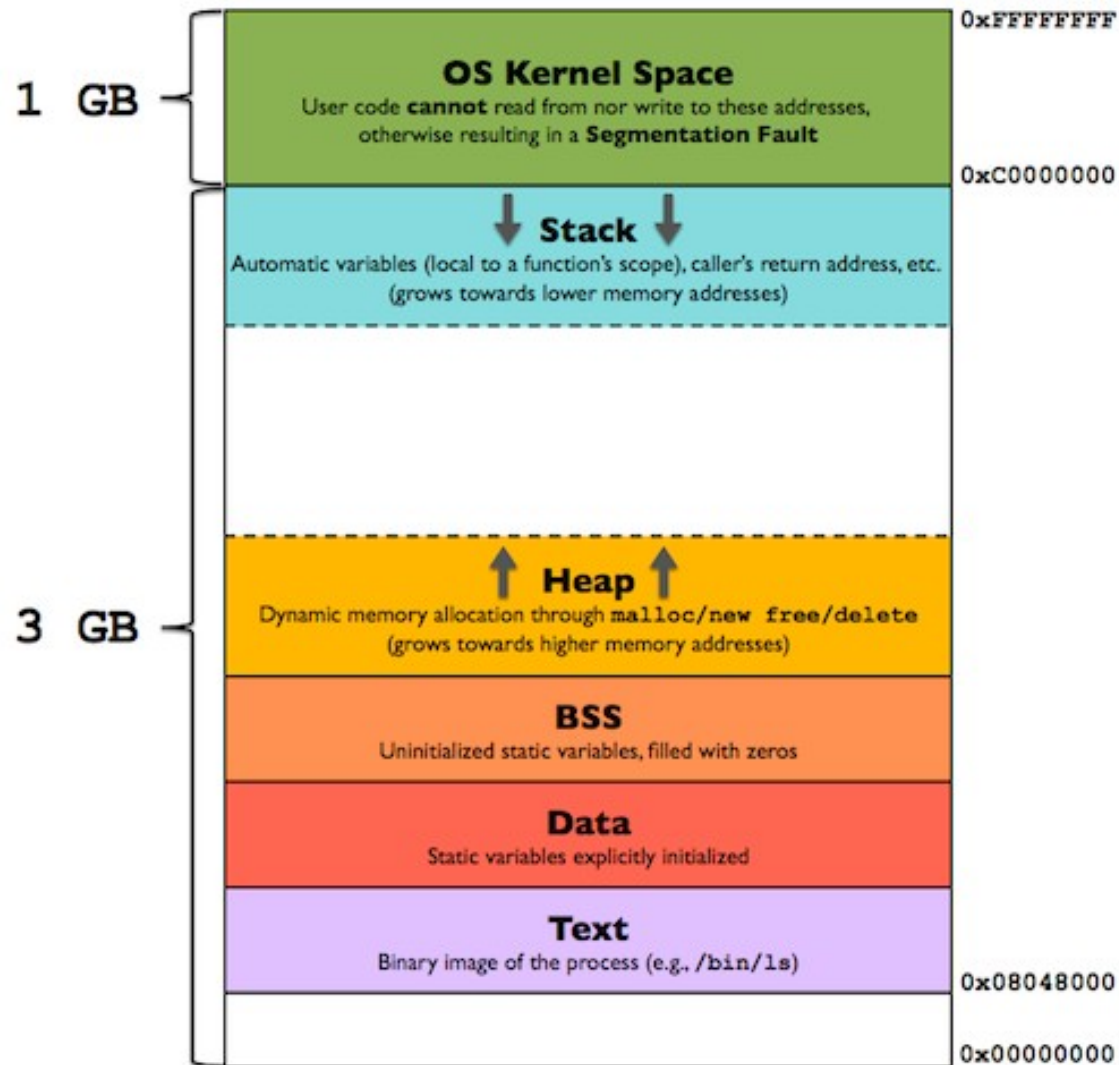
Segmentos da memória

- Diferentes representações



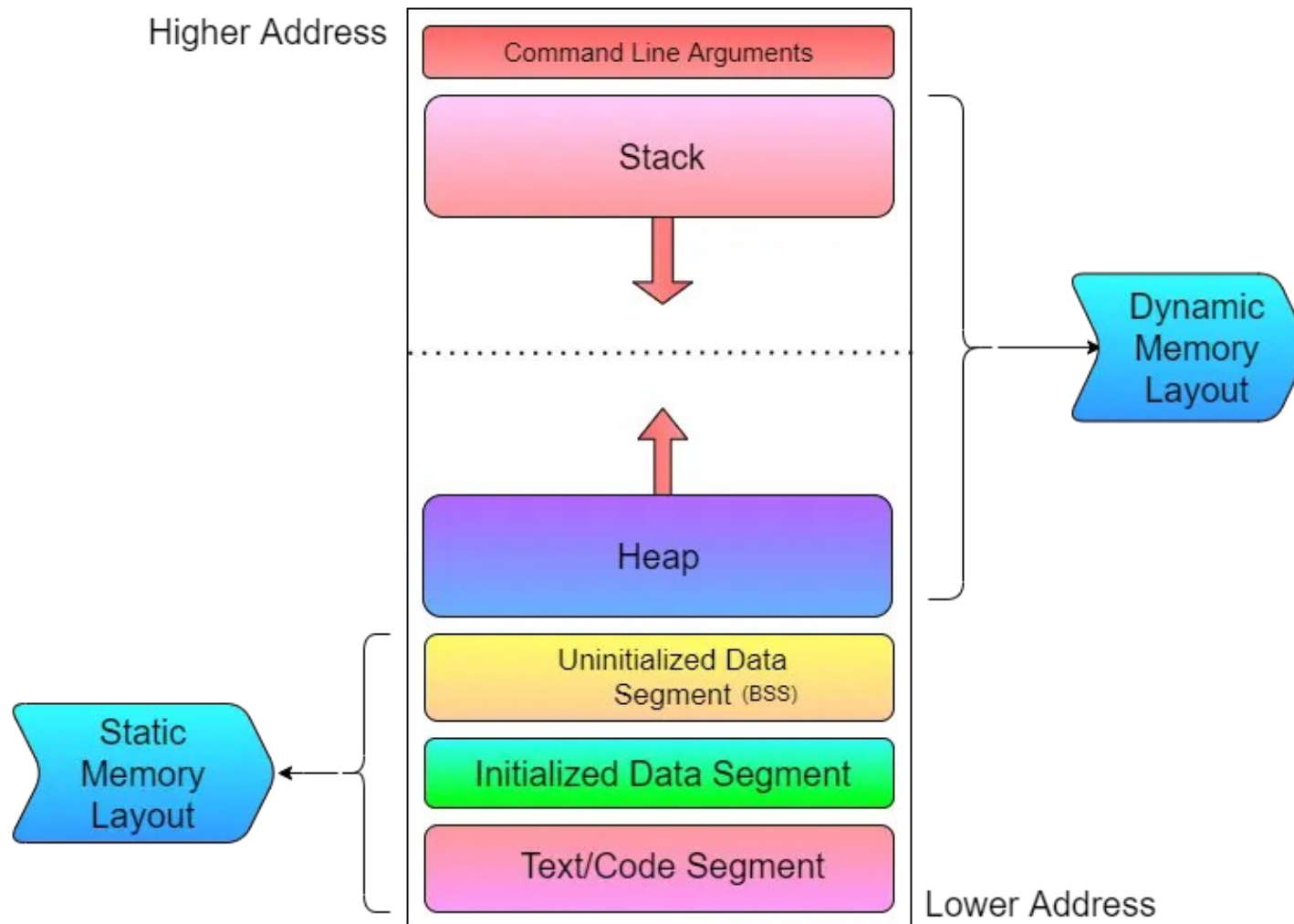
Segmentos da memória

- Diferentes representações



Segmentos da memória

- Diferentes representações



Entendendo:

- Cada programa em execução (processo) tem seu próprio layout de memória (user space) separado de outros programas.
- O espaço de memória é dividido em segmentos, cada um com uma função específica.
- Entender esses segmentos é fundamental para entender o funcionamento das variáveis nas linguagens de programação.
- Vamos ENTENDER e FAZER NA PRÁTICA.

Entendendo: TEXT

- **Text Segment (Code Segment):**
 - Tamanho fixo
 - Instruções executáveis do programa, o código sendo executado, em linguagem de máquina
 - Inclui a “lógica” do programa
 - Usualmente compartilhado
 - Geralmente:
 - Read-only
 - Execute

Praticando: TEXT

- Verifique o conteúdo do arquivo codigo01.c
- Compile o programa
- Utilize o comando size para verificar a área TEXT

```
[abrantestasf@ideapad ~/memoria]$ gcc -o codigo01 codigo01.c
```

```
[abrantestasf@ideapad ~/memoria]$ size codigo01
text      data      bss      dec      hex filename
1418      544       8        1970     7b2  codigo01
```

- Use o objdump para obter o segmento TEXT

```
[abrantestasf@ideapad ~/memoria]$ objdump -S codigo01
```

Entendendo: DATA

- **Data Segment (Initialized Data Segment):**
 - Variáveis globais, estáticas, constantes e extern que foram inicializadas com algum valor diferente de 0
 - Parte read-only e parte read-write
 - Alocado no início do programa
 - Tamanho fixo após alocação
 - Não compartilhado

Praticando: DATA

- Verifique o conteúdo dos arquivos código02a.c e código02b.c
- Compile os programas
- Utilize o comando `size` para verificar a área DATA

```
[abrantestasf@ideapad ~/memoria]$ size código02a
text      data      bss       dec       hex filename
1418      564       4         1986      7c2 código02a
```

```
[abrantestasf@ideapad ~/memoria]$ size código02b
text      data      bss       dec       hex filename
1454      552       8         2014      7de código02b
```

Entendendo: BSS

- **BSS (Block Started by Symbol, Uninitialized Data Segment):**
 - Variáveis globais, estáticas e extern que não foram inicializadas ou que foram inicializadas com 0
 - Se não inicializadas, o kernel inicializa com 0
 - Read-write
 - Alocado no início do programa
 - Tamanho fixo após alocação
 - Não compartilhado

Praticando: BSS

- Sua vez! Crie um programa em C para mostrar que variáveis globais e estáticas não inicializadas são armazenadas no segmento BSS!
- Utilize o comando `size` para ter certeza.

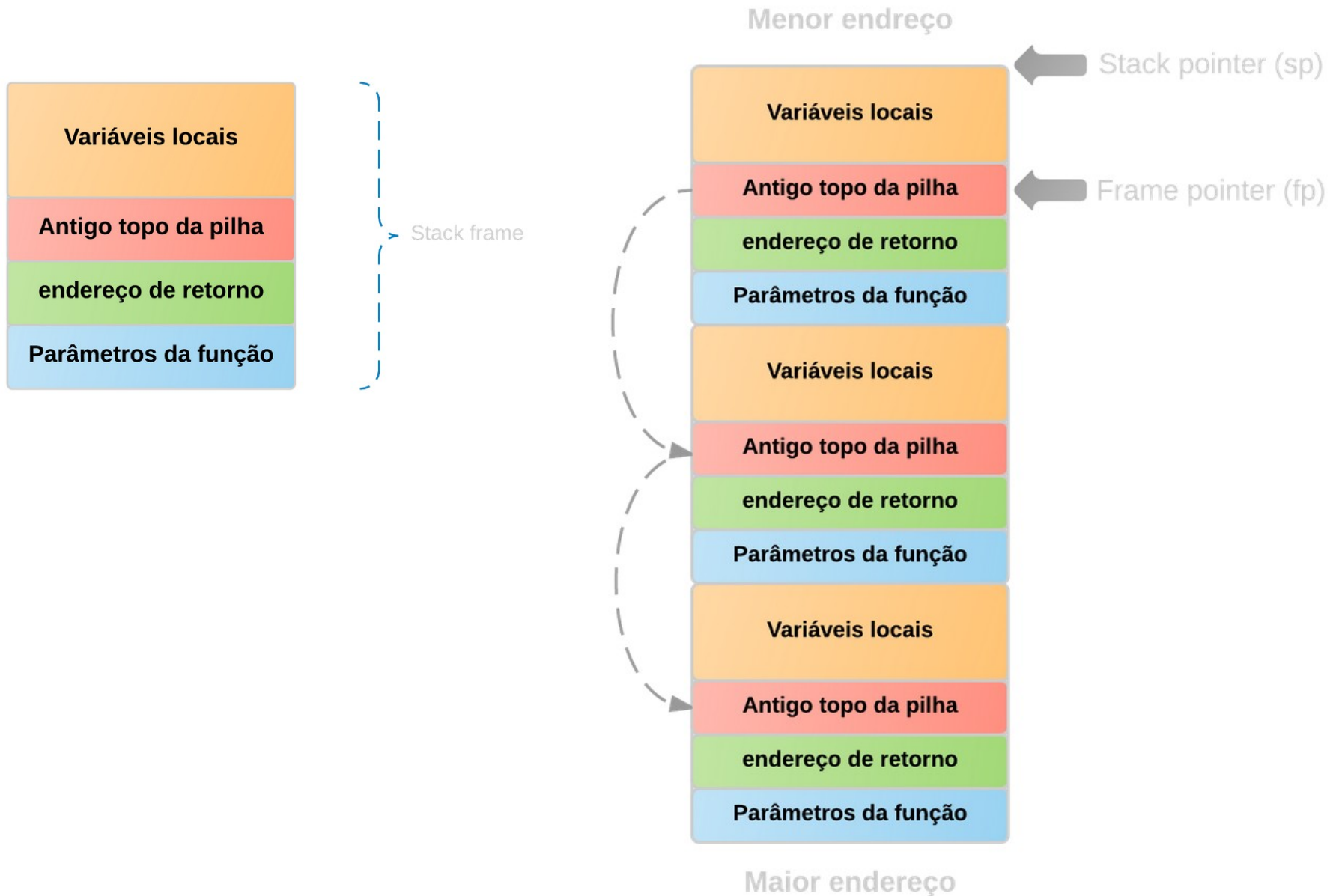
Entendendo: STACK

- **STACK (pilha de execução):**
 - É uma estrutura de dados do tipo pilha (LIFO)
 - Armazenamento temporário
 - Tudo que é necessário em uma chamada de função (incluindo funções recursivas): STACK FRAME
 - Função chamada e endereço de retorno
 - Parâmetros com seus argumentos
 - Variáveis locais

Entendendo: STACK

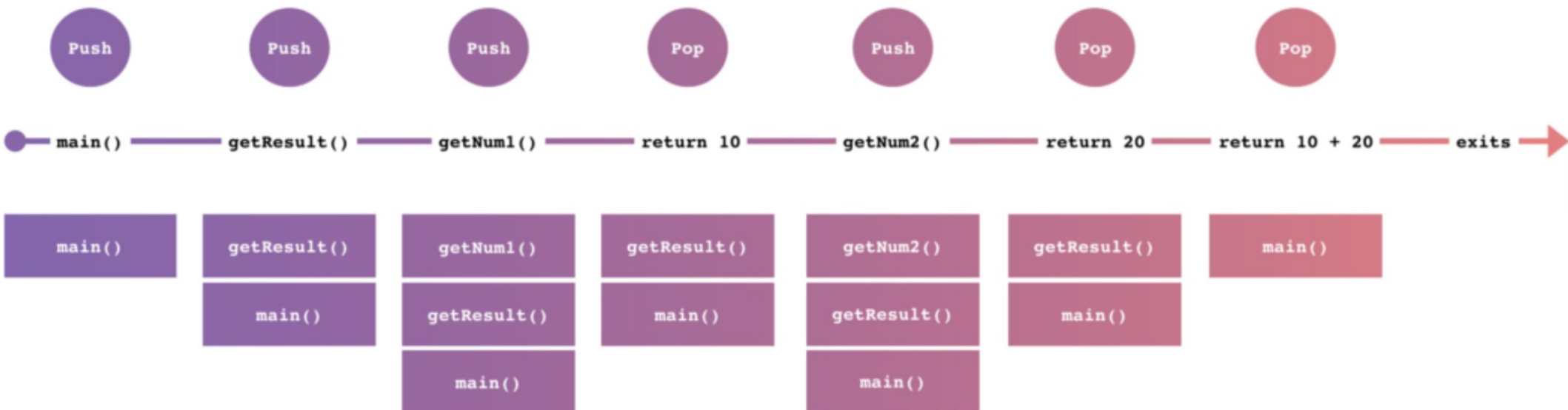
- **STACK (pilha de execução):**
 - Normalmente é **pequena!** 8 MB ou menos!
 - “ulimit -s” ou “ulimit -a”
 - Objetos grandes não devem ser colocados no stack, risco de stack overflow
 - Não é possível expandir “slots” no stack
 - Não é possível inserir/remover no meio (LIFO)
 - Alocação/desalocação é automática e rápida
 - O stack frame é descartado após a função retornar

Entendendo: STACK



Entendendo: STACK

```
int main() {  
    int result = getResult();  
}  
  
int getResult() {  
    int num1 = getNum1();  
    int num2 = getNum2();  
    return num1 + num2;  
}  
  
int getNum1() {  
    return 10;  
}  
  
int getNum2() {  
    return 20;  
}
```



Entendendo: STACK

Legend

type name value

Valid stack memory

type name value

Invalid stack memory



```
int hello() {
    int a = 100;
    return a;
}
int main() {
    int a;
    int b = -3;
    int c = 12345;
    int *p = &b;
    int d = hello();
    return 0;
}
```


Entendendo: STACK

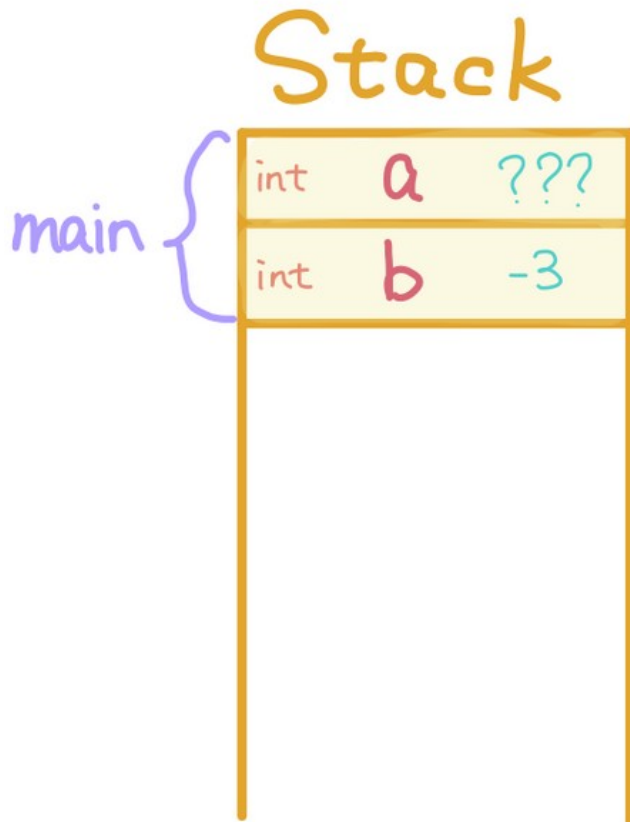
Legend

type name value

Valid stack memory

type name value

Invalid stack memory



```
int hello() {
    int a = 100;
    return a;
}
int main() {
    int a;
    int b = -3;
    int c = 12345;
    int *p = &b;
    int d = hello();
    return 0;
}
```

Entendendo: STACK

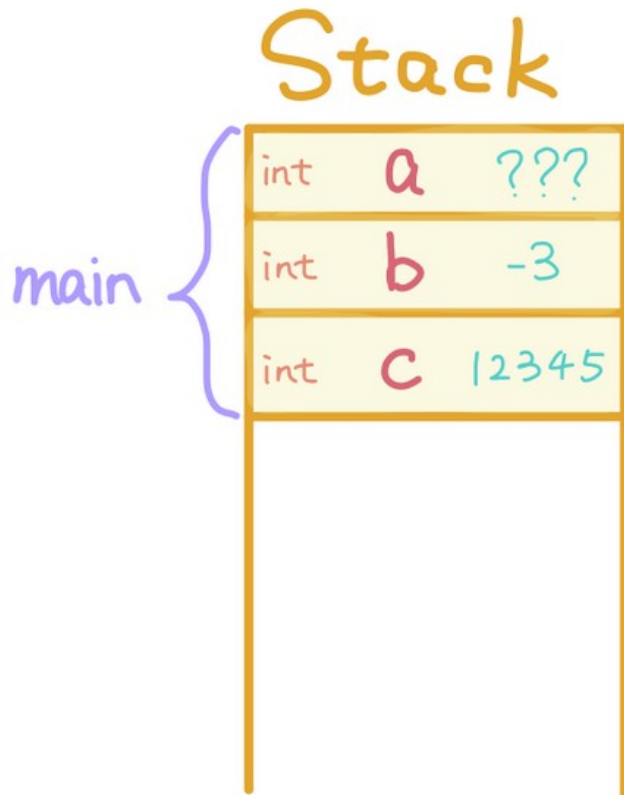
Legend

type name value

Valid stack memory

type name value

Invalid stack memory



```
int hello() {  
    int a = 100;  
    return a;  
}  
int main() {  
    int a;  
    int b = -3;  
    int c = 12345;  
    int *p = &b;  
    int d = hello();  
    return 0;  
}
```

Entendendo: STACK

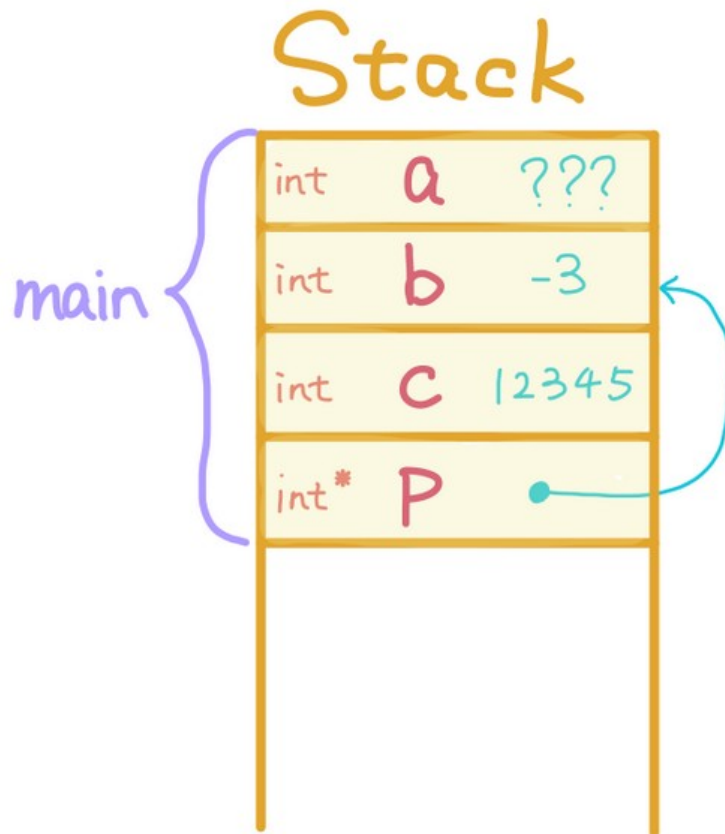
Legend

type name value

Valid stack memory

type name value

Invalid stack memory



```
int hello() {
    int a = 100;
    return a;
}
int main() {
    int a;
    int b = -3;
    int c = 12345;
    int *p = &b;
    int d = hello();
    return 0;
}
```

Entendendo: STACK

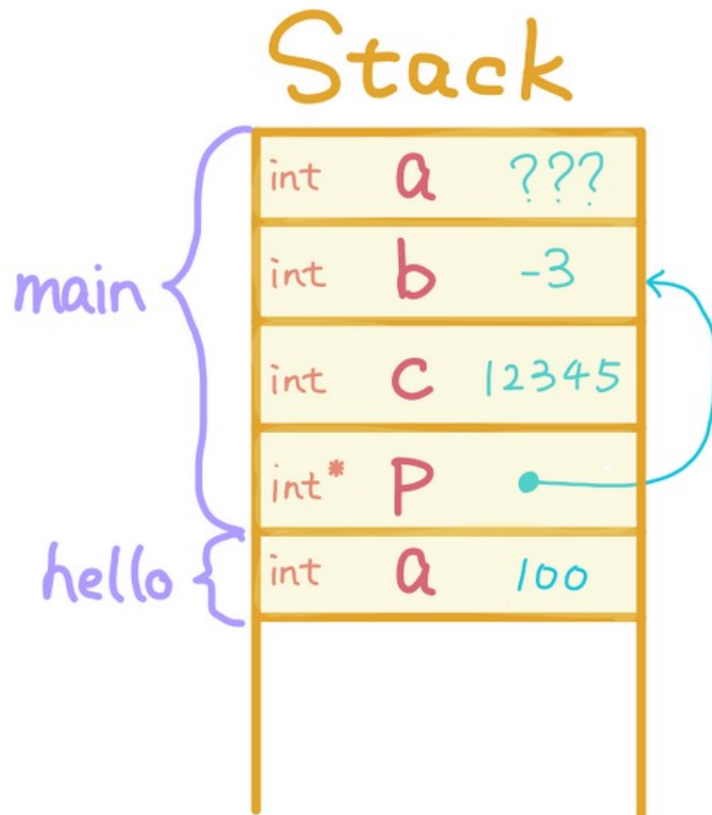
Legend

type name value

Valid stack memory

type name value

Invalid stack memory



```
int hello() {  
    int a = 100;  
    return a;  
}  
int main() {  
    int a;  
    int b = -3;  
    int c = 12345;  
    int *p = &b;  
    int d = hello();  
    return 0;  
}
```

Entendendo: STACK

Legend

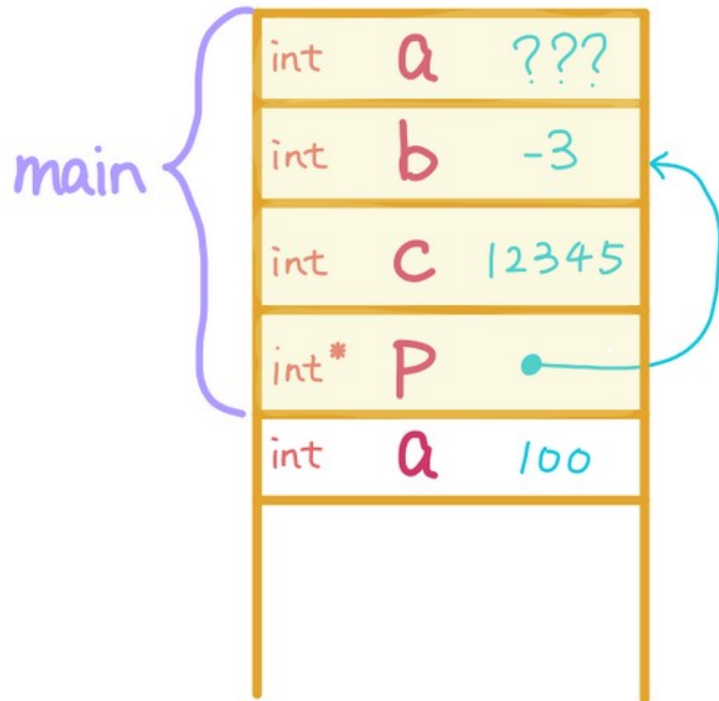
type name value

Valid stack memory

type name value

Invalid stack memory

Stack



```
int hello() {  
    int a = 100;  
    return a;  
}  
int main() {  
    int a;  
    int b = -3;  
    int c = 12345;  
    int *p = &b;  
    int d = hello();  
    return 0;  
}
```

Entendendo: STACK

Legend

type name value

Valid stack memory

type name value

Invalid stack memory

Stack

int	a	???
int	b	-3
int	c	12345
int*	p	•
int	d	100

main

```
int hello() {  
    int a = 100;  
    return a;  
}  
int main() {  
    int a;  
    int b = -3;  
    int c = 12345;  
    int *p = &b;  
    int d = hello();  
    return 0;  
}
```

Entendendo: STACK

Legend

type name value

Valid stack memory

type name value

Invalid stack memory

Stack

int	a	???
int	b	-3
int	c	12345
int*	p	•
int	d	100

```
int hello() {  
    int a = 100;  
    return a;  
}  
int main() {  
    int a;  
    int b = -3;  
    int c = 12345;  
    int *p = &b;  
    int d = hello();  
    return 0;  
}
```

Entendendo: STACK

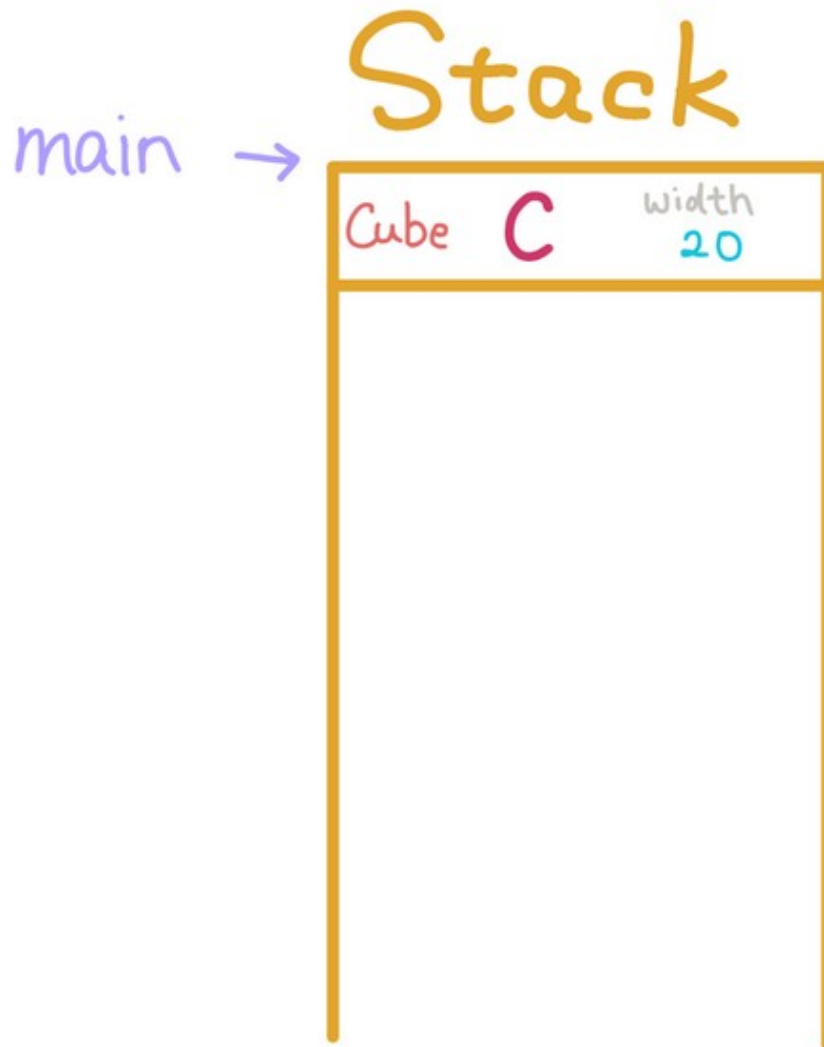
- **STACK (pilha de execução):**
 - Cuidado para não tentar usar uma variável criada no stack FORA do escopo da função que foi chamada!
 - Lembre-se que a desalocação é automática!!!
 - Fonte comum de erros nos programas em C, C++

Entendendo: STACK



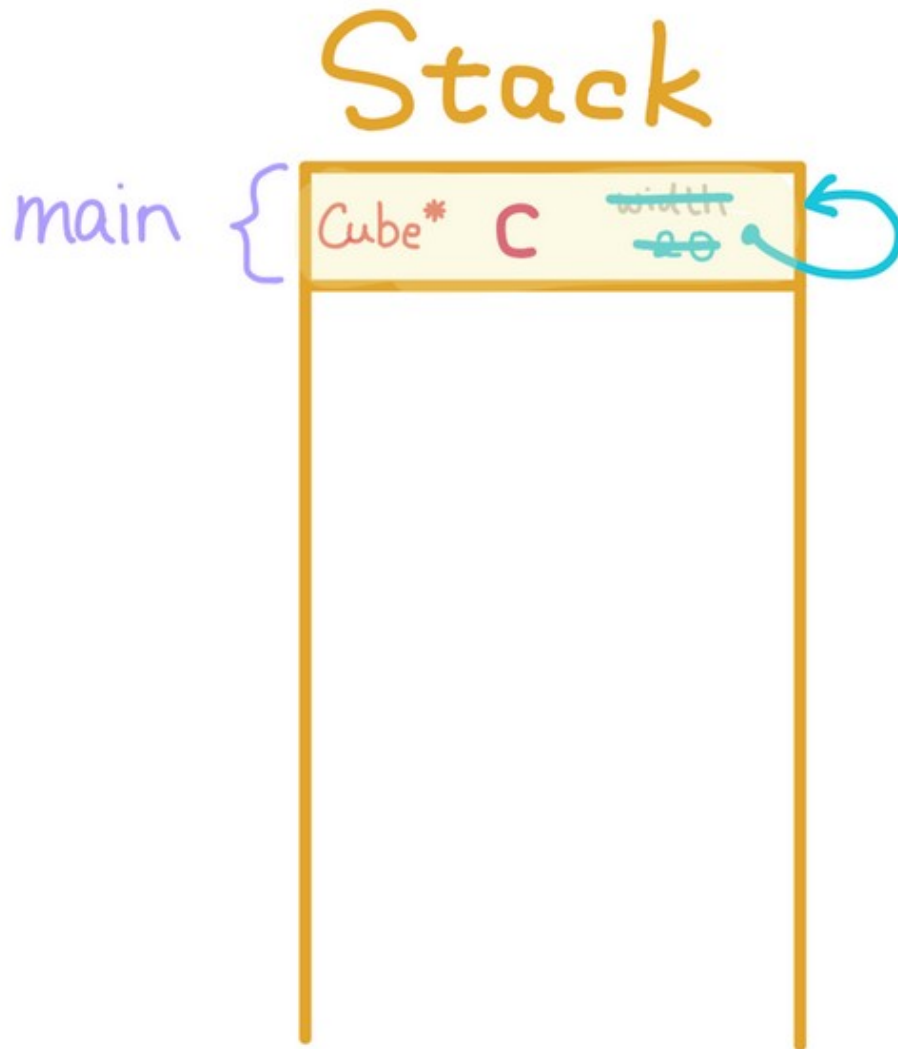
```
Cube *CreateCube() {  
    Cube c(20);  
    return &c;  
}  
  
int main() {  
    Cube *c = CreateCube();  
    double r = c->getVolume();  
    double v = c->getSurfaceArea();  
    return 0;  
}
```

Entendendo: STACK



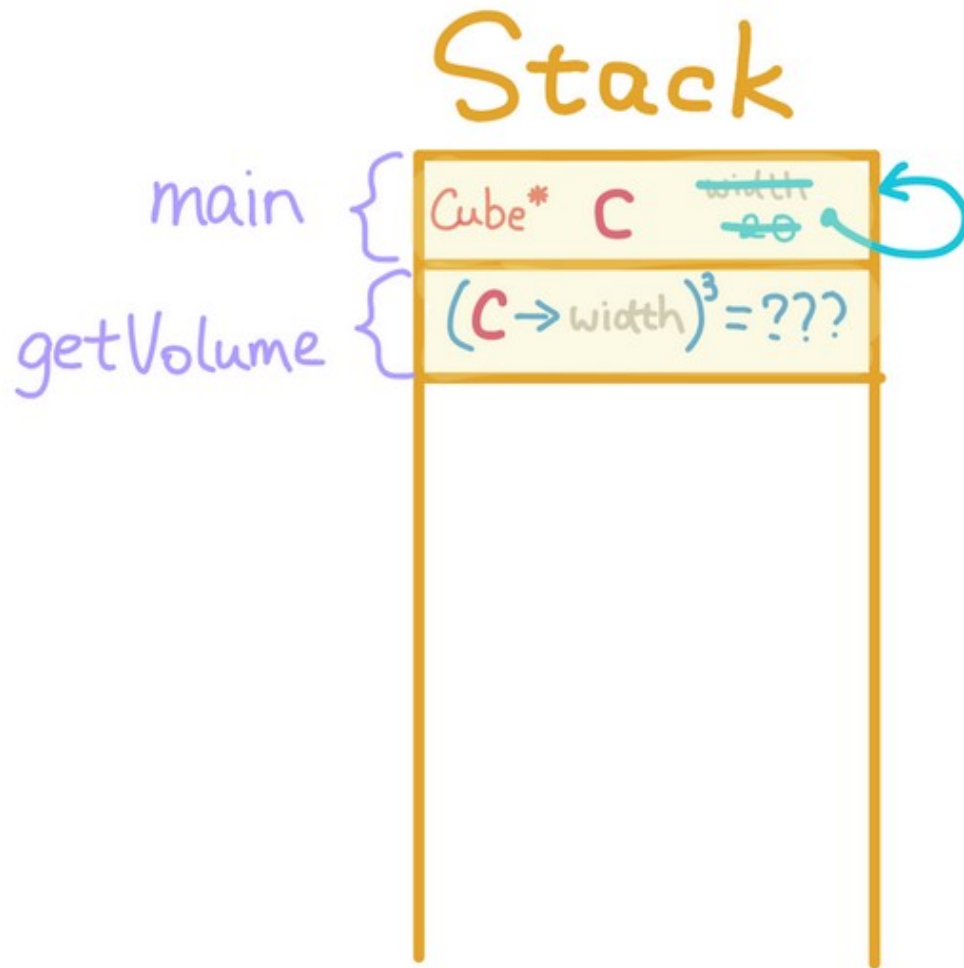
```
Cube *CreateCube() {  
    Cube c(20);  
    return &c;  
}  
  
int main() {  
    Cube *c = CreateCube();  
    double r = c->getVolume();  
    double v = c->getSurfaceArea();  
    return 0;  
}
```

Entendendo: STACK



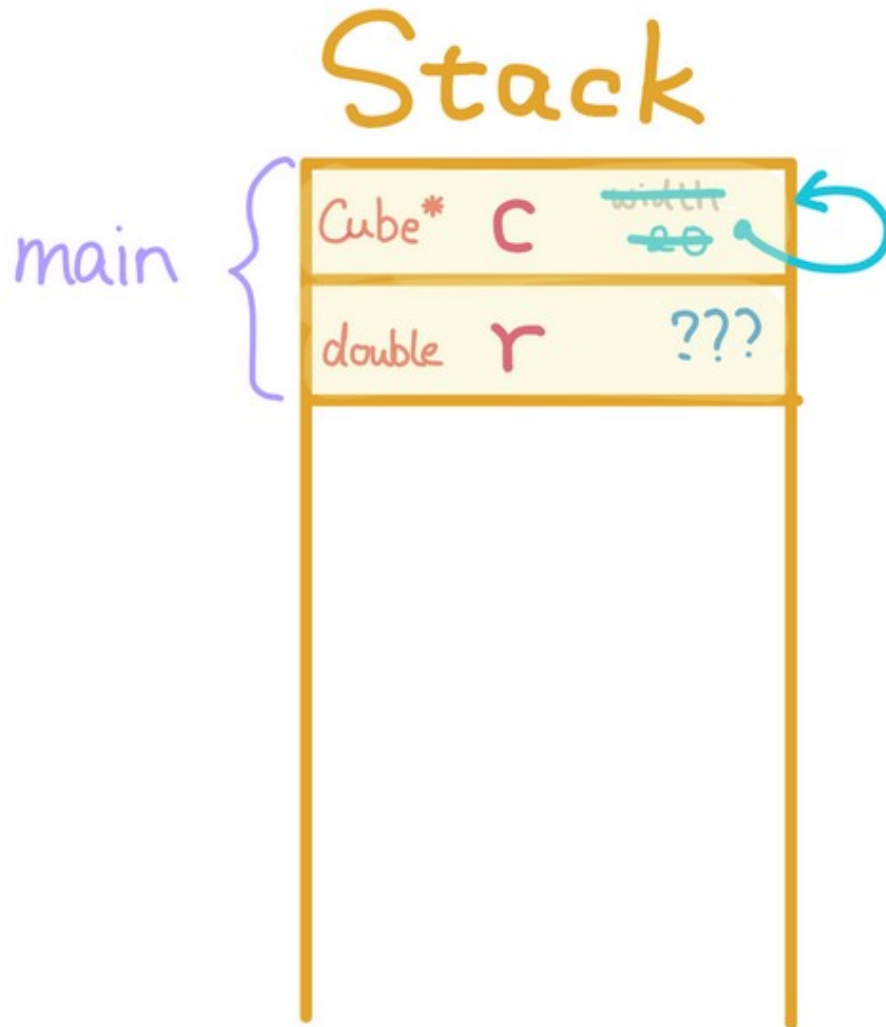
```
Cube *CreateCube() {  
    Cube c(20);  
    return &c;  
}  
  
int main() {  
    Cube *c = CreateCube();  
    double r = c->getVolume();  
    double v = c->getSurfaceArea();  
    return 0;  
}
```

Entendendo: STACK



```
Cube *CreateCube() {  
    Cube c(20);  
    return &c;  
}  
  
int main() {  
    Cube *c = CreateCube();  
    double r = c->getVolume();  
    double v = c->getSurfaceArea();  
    return 0;  
}
```

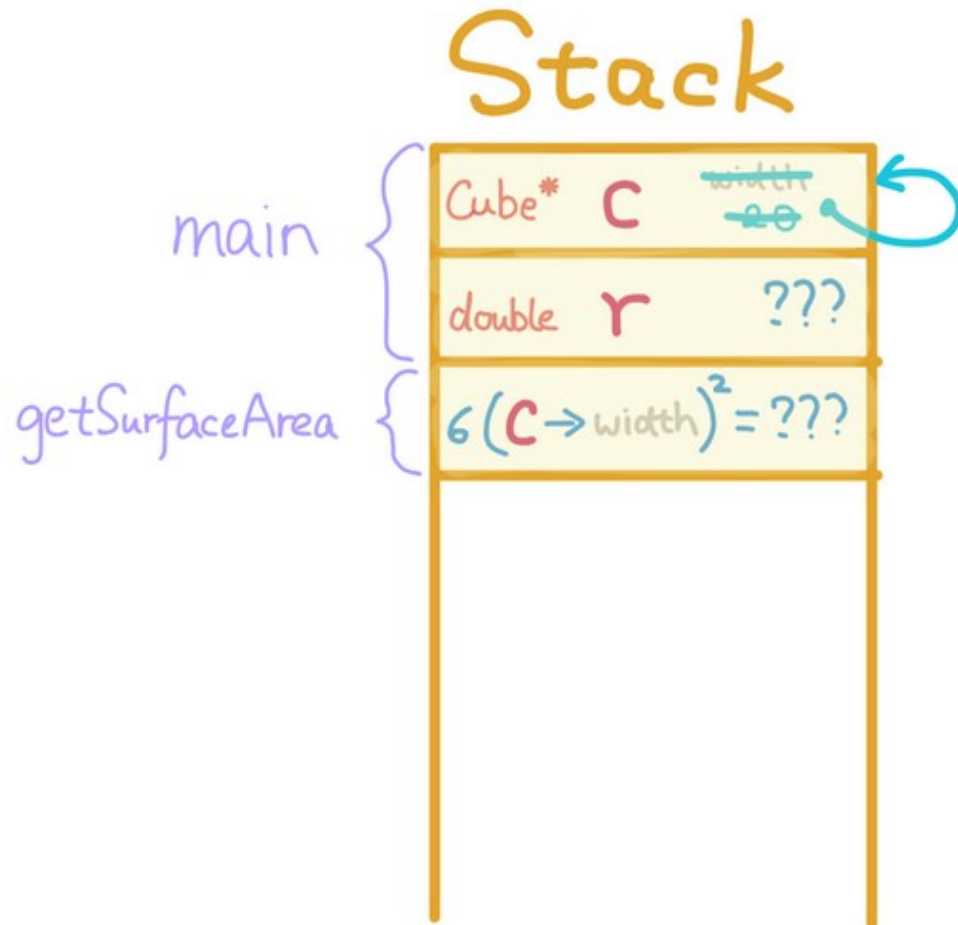
Entendendo: STACK



```
Cube *CreateCube() {
    Cube c(20);
    return &c;
}

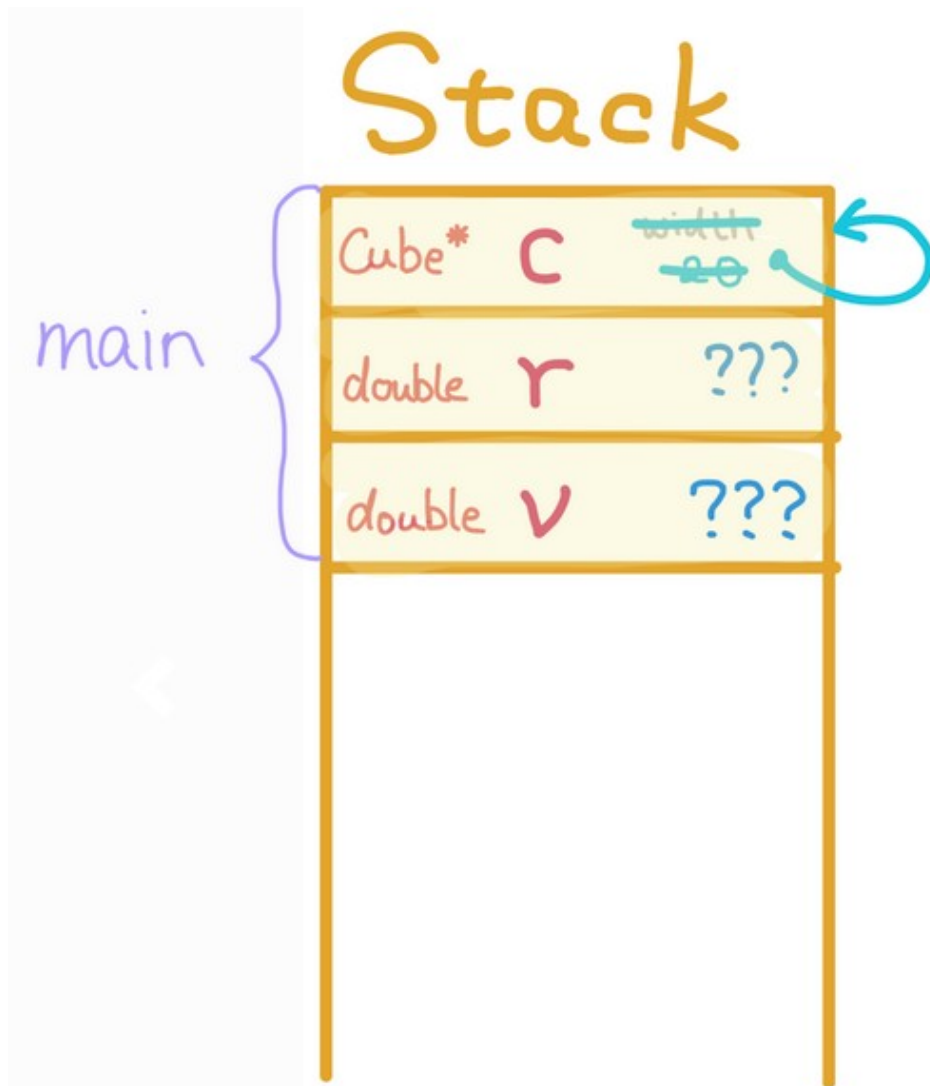
int main() {
    Cube *c = CreateCube();
    double r = c->getVolume();
    double v = c->getSurfaceArea();
    return 0;
}
```

Entendendo: STACK



```
Cube *CreateCube() {  
    Cube c(20);  
    return &c;  
}  
  
int main() {  
    Cube *c = CreateCube();  
    double r = c->getVolume();  
    double v = c->getSurfaceArea();  
    return 0;  
}
```

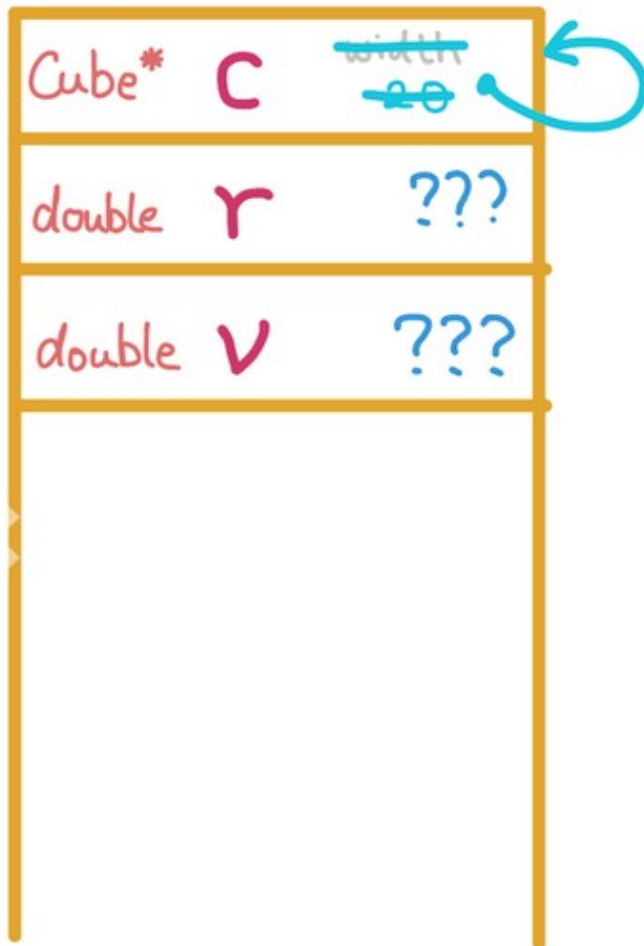
Entendendo: STACK



```
Cube *CreateCube() {  
    Cube c(20);  
    return &c;  
}  
  
int main() {  
    Cube *c = CreateCube();  
    double r = c->getVolume();  
    double v = c->getSurfaceArea();  
    return 0;  
}
```

Entendendo: STACK

Stack



```
Cube *CreateCube() {  
    Cube c(20);  
    return &c;  
}  
  
int main() {  
    Cube *c = CreateCube();  
    double r = c->getVolume();  
    double v = c->getSurfaceArea();  
    return 0;  
}
```


Praticando: STACK

- Vamos ver os endereços do STACK de um processo!

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

int main(int argc, char **argv)
{
    // Cria algumas variáveis para vermos o endereço:
    int variavel_local_inicializada = 78;
    int variavel_local_nao_inicializada;
    printf("Endereço de: variavel_local_inicializada      = %p\n", &variavel_local_inicializada);
    printf("Endereço de: variavel_local_nao_inicializada = %p\n", &variavel_local_nao_inicializada);

    // Cria um loop infinito para dar tempo de verificarmos
    // o mapa de endereços de memória em /proc/<ID do processo>,
    // em outro terminal
    while(1);

    // Retorna
    return 0;

    // ATENÇÃO! Não se esqueça de MATAR MANUALMENTE o processo deste
    // programa, ou ele continuará rodando infinitamente!
}
```

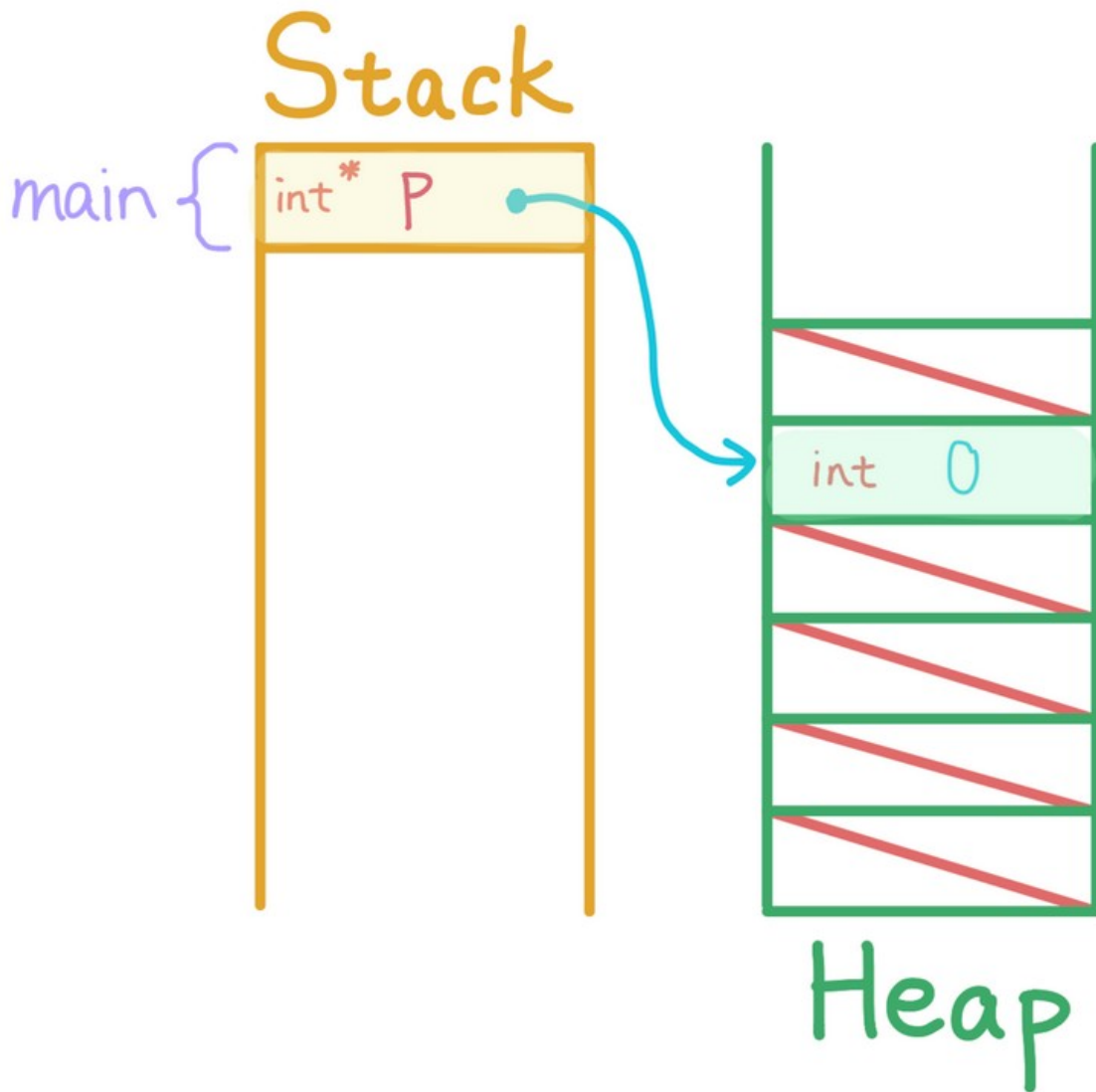
Entendendo: HEAP

- **HEAP (“montão”, “coleção”, “um monte”):**
 - Alocação dinâmica das variáveis cujo tamanho só é conhecido em runtime (não podem ser determinadas pelo compilador)
 - Mais permanente do que o stack
 - Alocada/Desalocada de forma explícita! Sem gerenciamento automático!
 - Se esquecer de desalocar, é memory leak (exceto se a linguagem tiver garbage collector).
 - O escopo não é limitado (variáveis podem ser referenciadas de diversos locais)

Entendendo: HEAP

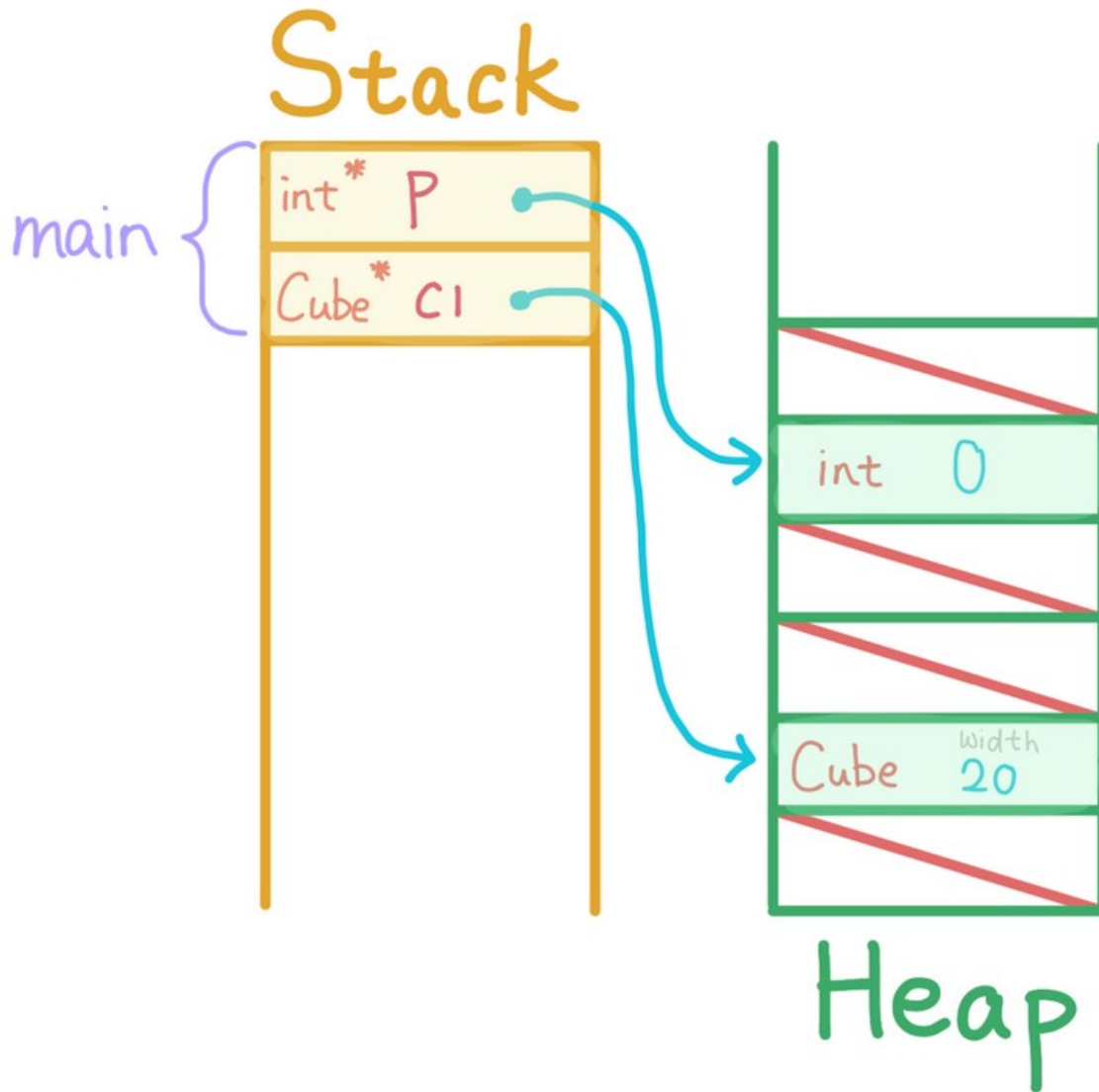
- **HEAP (“montão”, “coleção”, “um monte”):**
 - É bem grande! Padrão do gcc: 4 GB
 - É possível aumentar com flags específicas durante a compilação (mcmmodel = large)
 - Não é uma pilha, pode ser acessada, alocada e desalocada em diferentes posições
 - Armazena tudo o que não pertence ao TEXT, DATA, BSS e STACK.
 - Não organiza nada, é meio “bagunçado”

Entendendo: HEAP



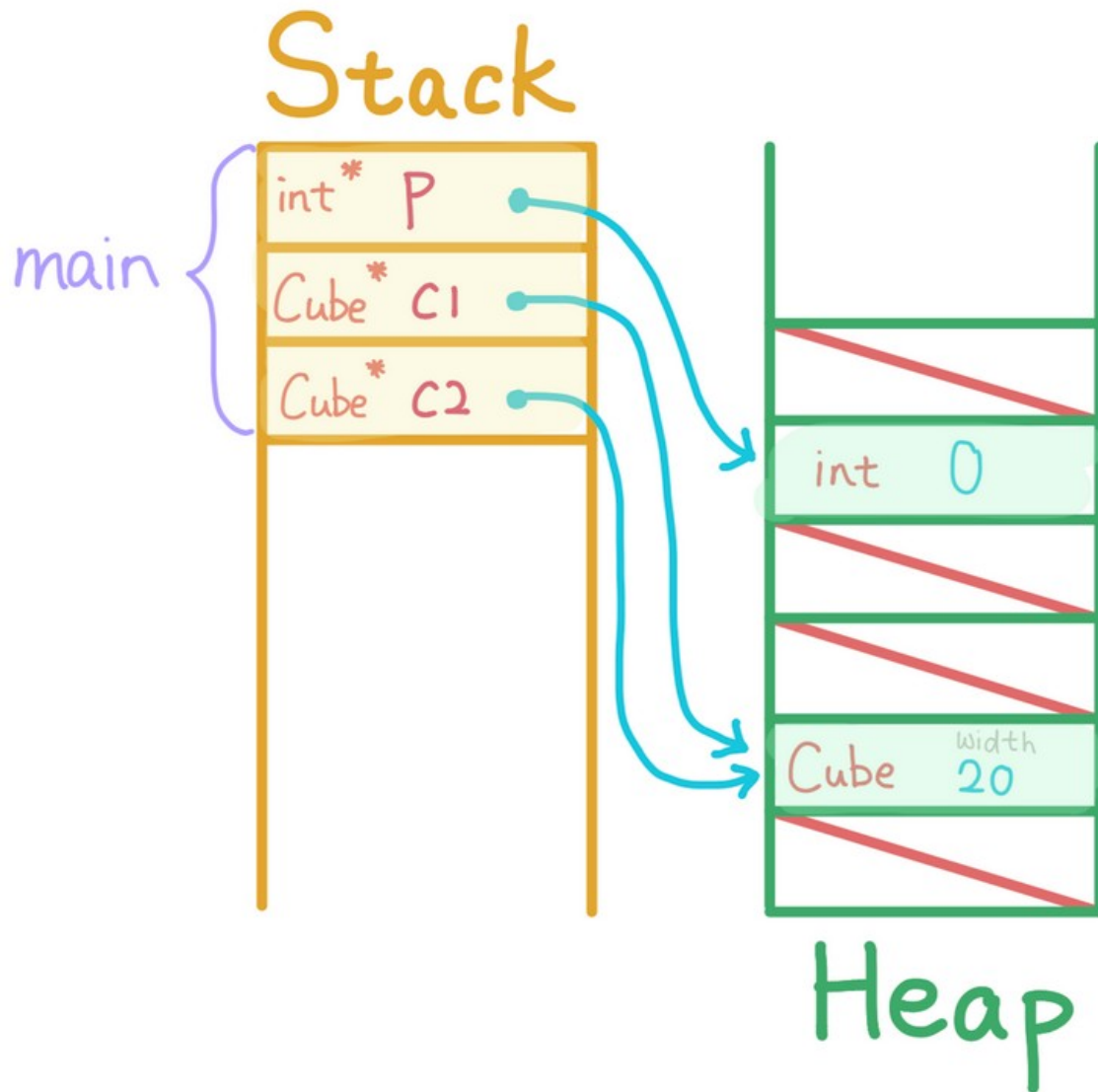
```
int main() {  
    int *p = new int;  
    Cube *c1 = new Cube();  
    Cube *c2 = c1;  
    c2->setLength( 10 );  
    return 0;  
}
```

Entendendo: HEAP



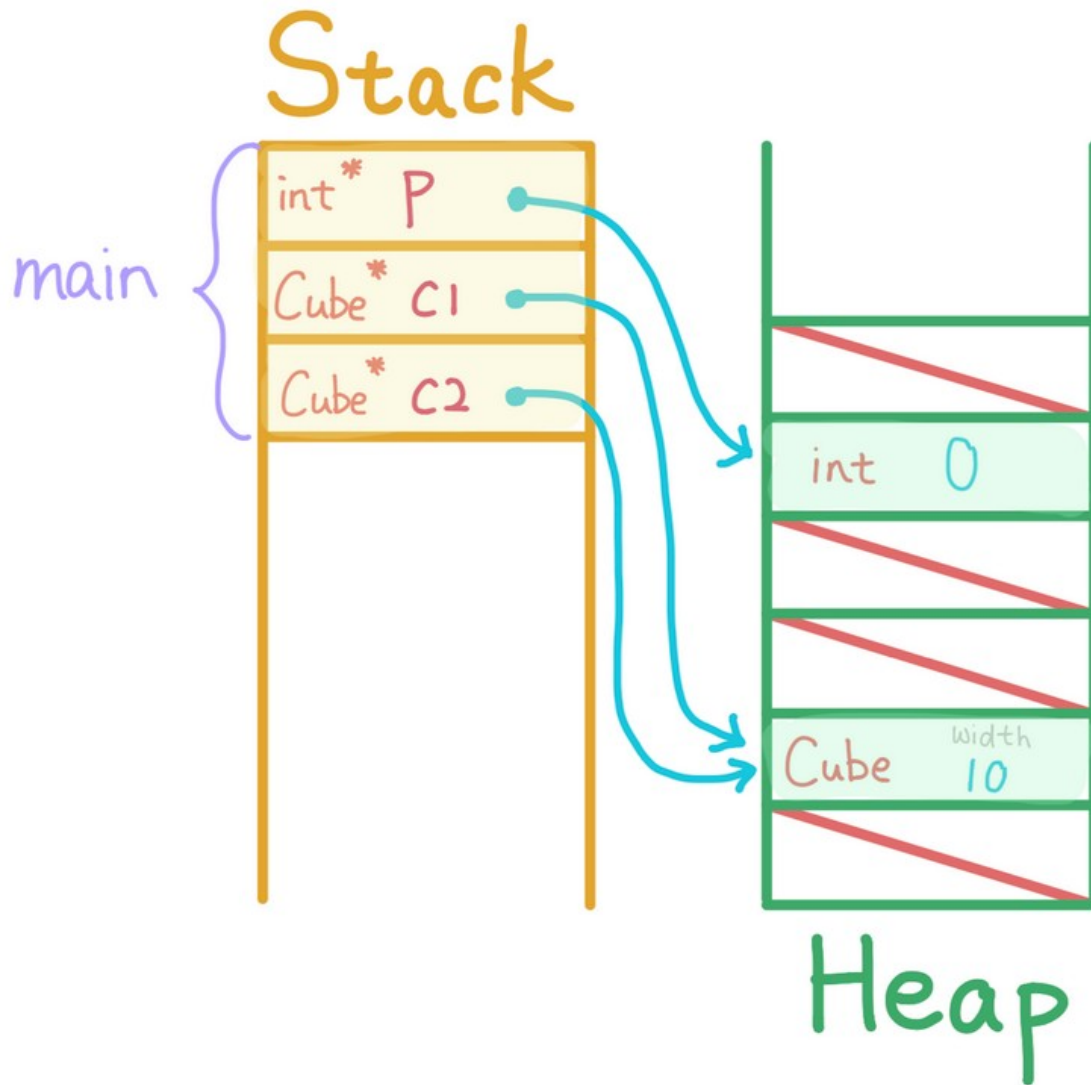
```
int main() {  
    int *p = new int;  
    Cube *c1 = new Cube();  
    Cube *c2 = c1;  
    c2->setLength( 10 );  
    return 0;  
}
```

Entendendo: HEAP



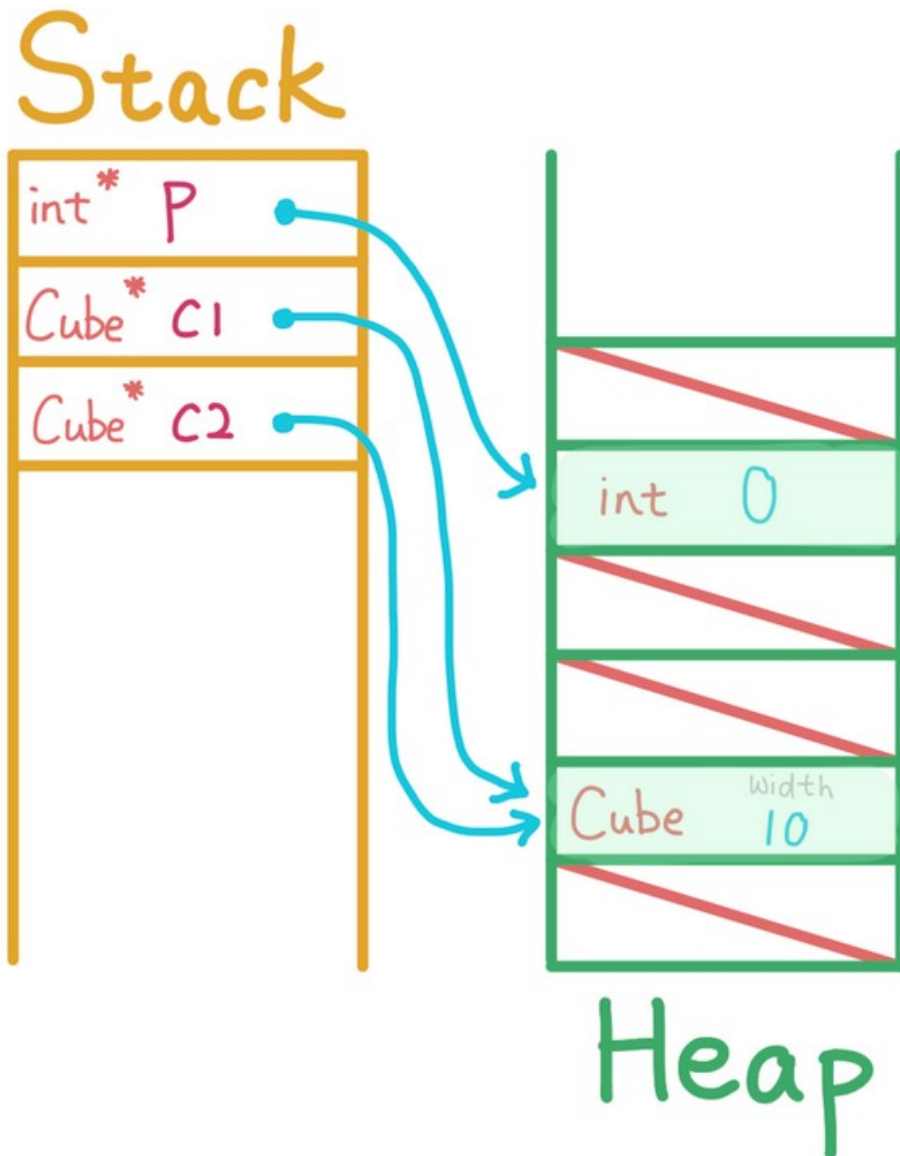
```
int main() {  
    int *p = new int;  
    Cube *c1 = new Cube();  
    Cube *c2 = c1;  
    c2->setLength( 10 );  
    return 0;  
}
```

Entendendo: HEAP



```
int main() {  
    int *p = new int;  
    Cube *c1 = new Cube();  
    Cube *c2 = c1;  
    c2->setLength( 10 );  
    return 0;  
}
```

Entendendo: HEAP



```
int main() {  
    int *p = new int;  
    Cube *c1 = new Cube();  
    Cube *c2 = c1;  
    c2->setLength( 10 );  
    return 0;  
}
```

Que erro o programador cometeu aqui?

Praticando: HEAP

- Vamos ver os endereços do HEAP de um processo!

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

int main(int argc, char **argv)
{
    // Cria algumas variáveis para vermos o endereço:
    int *memoria_para_int = malloc(sizeof(int));
    printf("Endereço de memória alocada para o int: = %p\n", memoria_para_int);

    // Cria um loop infinito para dar tempo de verificarmos
    // o mapa de endereços de memória em /proc/<ID do processo>,
    // em outro terminal
    while(1);

    // Retorna
    return 0;

    // ATENÇÃO! Não se esqueça de MATAR MANUALMENTE o processo deste
    // programa, ou ele continuará rodando infinitamente!
}
```

HEAP x STACK

- **Velocidade de acesso:**
 - STACK: rápido
 - HEAP: mais lento
- **Escopo:**
 - STACK: local
 - HEAP: global
- **Alocação/Desalocação:**
 - STACK: automática
 - HEAP: dinâmica (manual)

HEAP x STACK

- **Gerenciamento de memória:**
 - STACK: eficiente, não fragmentado
 - HEAP: menos eficiente, fragmentado
- **Limite:**
 - STACK: limitado e pequeno
 - HEAP: grande, praticamente “ilimitado”

Lembrete!

