Software Vulnerabilities

Software Vulnerabilities

1. Introduction
2. Program memory layout
3. Stack layout
4. Buffer overflow vulnerability

Software Vulnerabilities

1. Introduction
2. Program memory layout
3. Stack layout
4. Buffer overflow vulnerability

Common Weakness Enumeration

A Community-Developed List of Software & Hardware Weakness Types

1425 - Weaknesses in the 2023 CWE Top 25 Most Dangerous Software Weaknesses

- B Out-of-bounds Write - *(787)* **#1**
- B Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') -
- B Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'
- V Use After Free - *(416)*
- B Improper Neutralization of Special Elements used in an OS Command ('OS Command I
- C Improper Input Validation - *(20)*
- B Out-of-bounds Read - *(125)*
- B Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') - *(22)*
- Cross-Site Request Forgery (CSRF) - *(352)*
- B Unrestricted Upload of File with Dangerous Type - *(434)*
- C Missing Authorization - *(862)*
- B NULL Pointer Dereference - *(476)*
- C Improper Authentication - *(287)*
- B Integer Overflow or Wraparound - *(190)*

https://cwe.mitre.org/data/index.html

CWE-787: Out-of-bounds Write

Weakness ID: 787
Abstraction: Base
Structure: Simple

View customized information: Conceptual | Operational | Mapping Friendly | **Complete** | Custom

Description

The product writes data past the end, or before the beginning, of the intended buffer.

Extended Description

Typically, this can result in corruption of data, a crash, or code execution. The product may modify an index or perform pointer arithmetic that references a memory location that is outside of the boundaries of the buffer. A subsequent write operation then produces undefined or unexpected results.

Alternate Terms

Memory Corruption: Often used to describe the consequences of writing to memory outside the bounds of a buffer, or to memory that is invalid, when the root cause is something other than a sequential copy of excessive data from a fixed starting location. This may include issues such as incorrect pointer arithmetic, accessing invalid pointers due to incomplete initialization or memory release, etc.

Relationships

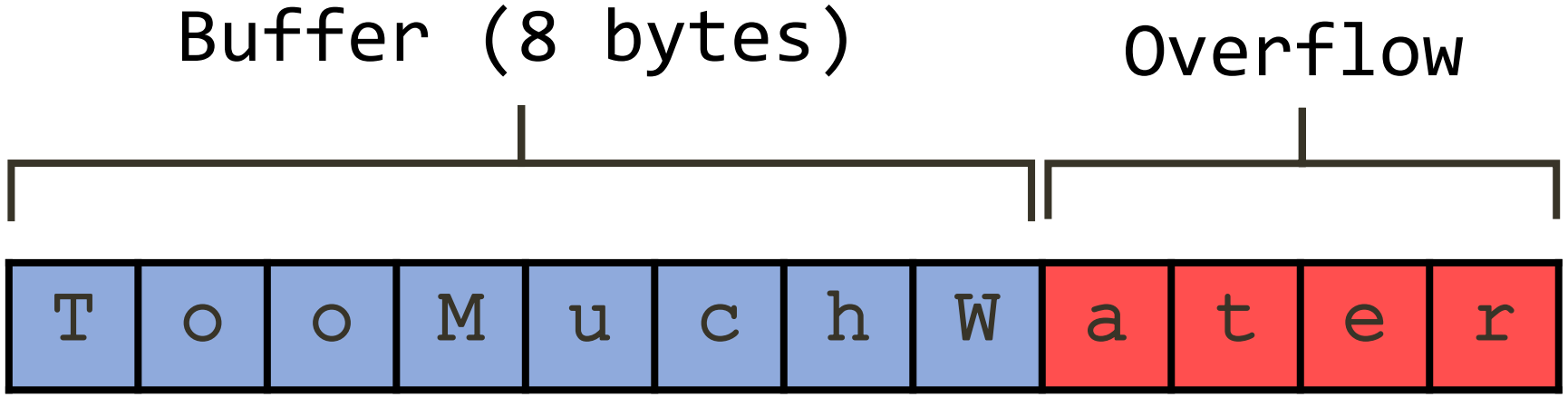
Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer
ParentOf	V	121	Stack-based Buffer Overflow
ParentOf	V	122	Heap-based Buffer Overflow
ParentOf	B	123	Write-what-where Condition
ParentOf	B	124	Buffer Underwrite ('Buffer Underflow')
CanFollow	B	822	Untrusted Pointer Dereference
CanFollow	B	823	Use of Out-of-range Pointer Offset
CanFollow	B	824	Access of Uninitialized Pointer
CanFollow	B	825	Expired Pointer Dereference

https://cwe.mitre.org/data/definitions/787.html

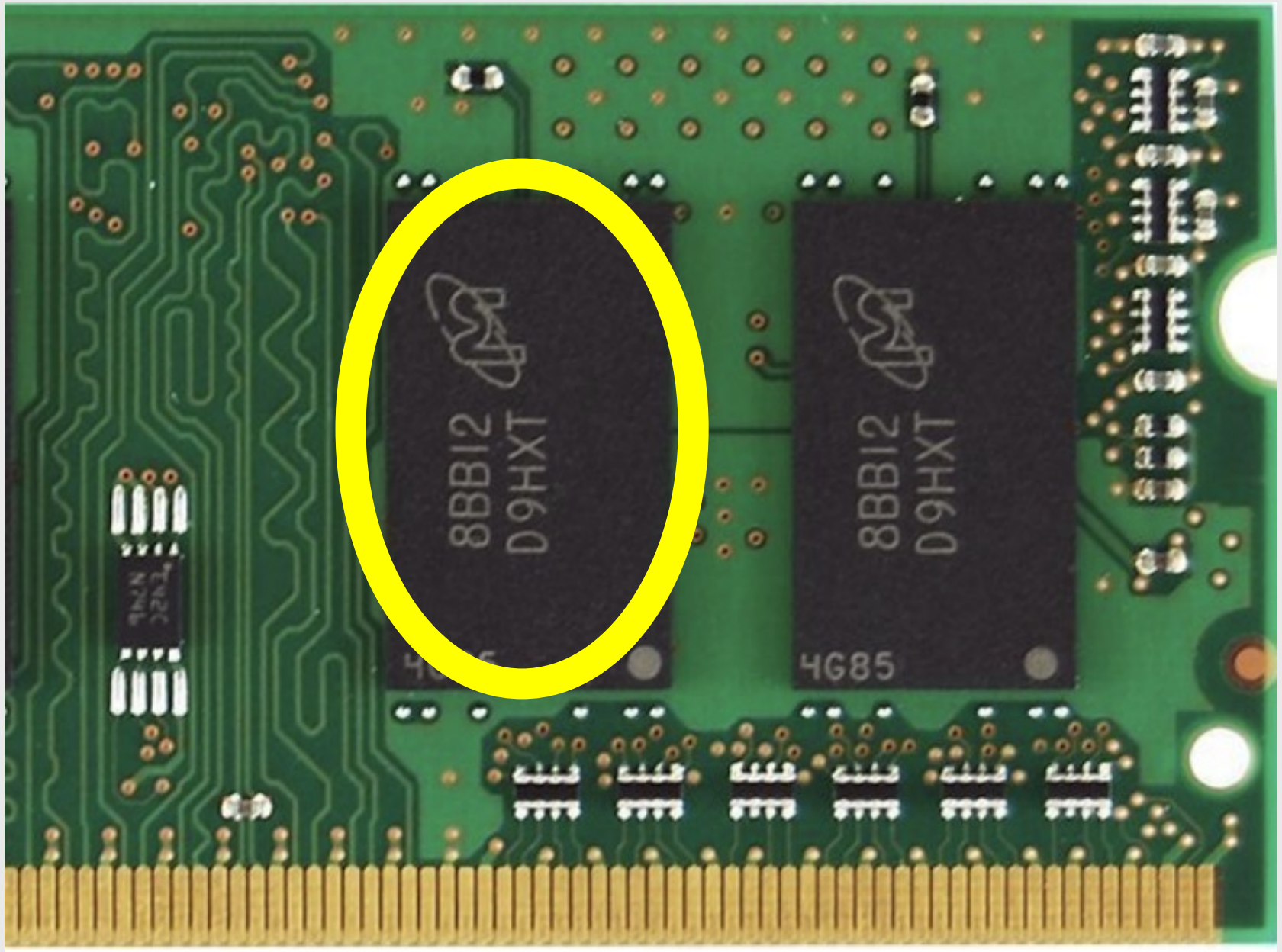
What is a buffer overflow?

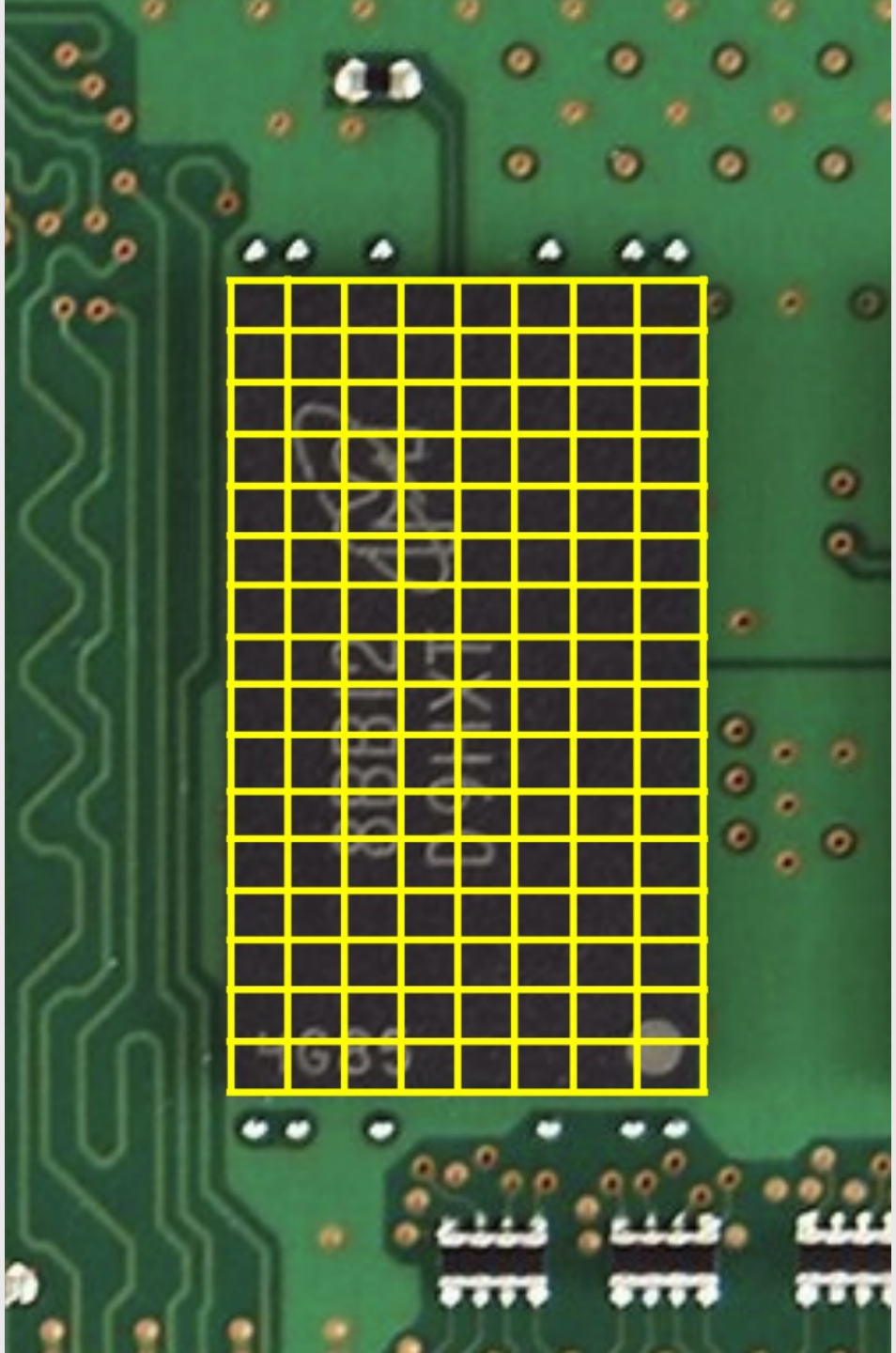
More bytes are written to a buffer than allocated for it



Software Vulnerabilities

1. Introduction
2. Program memory layout
3. Stack layout
4. Buffer overflow vulnerability

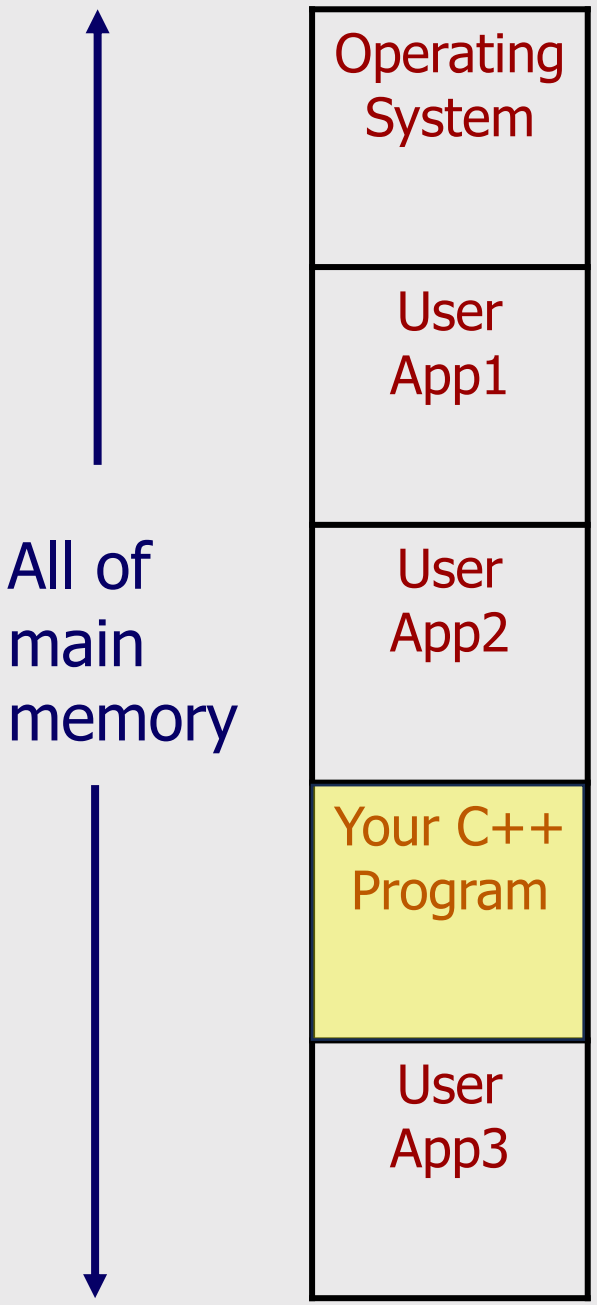


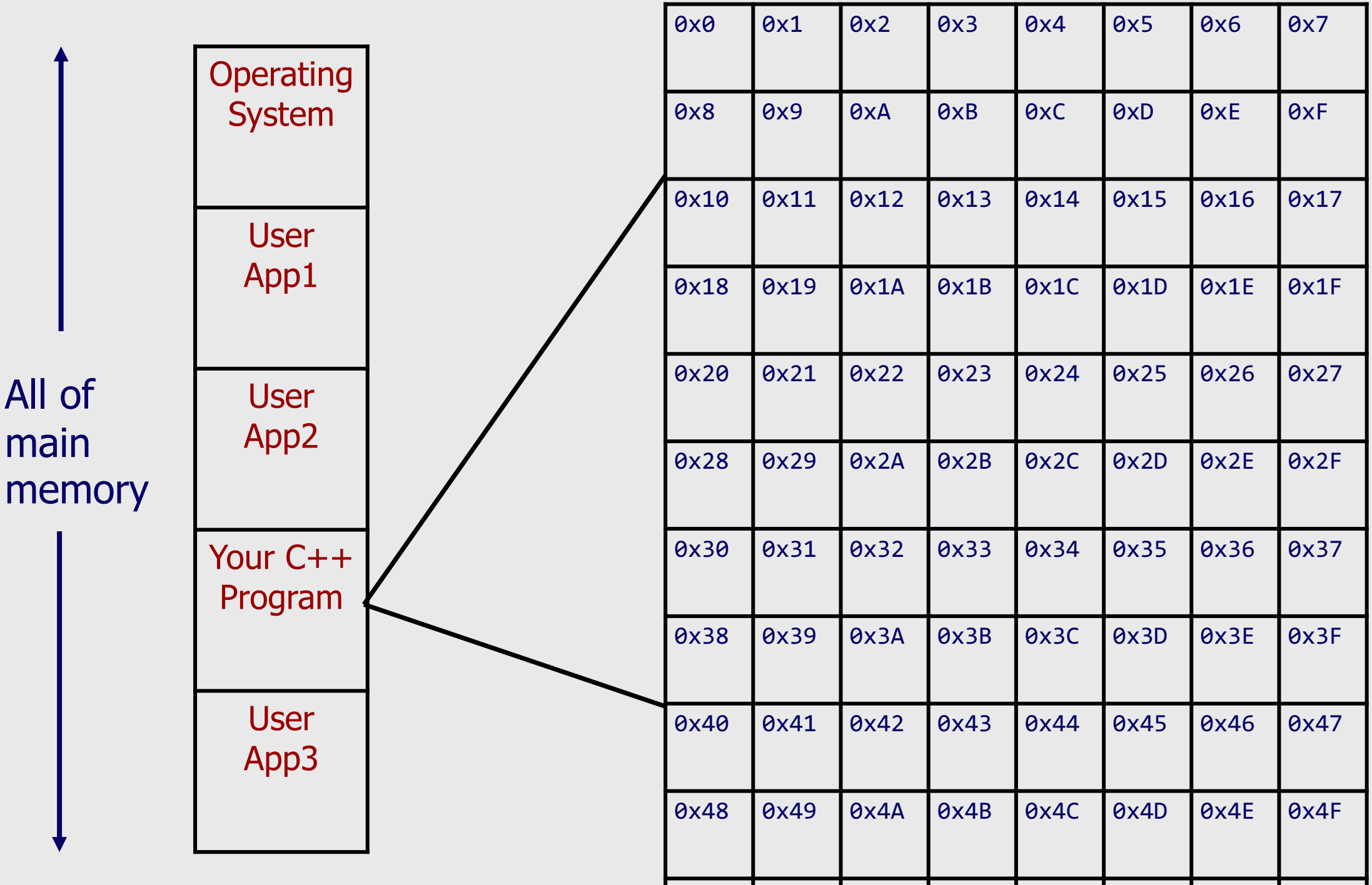


Low Memory Addresses

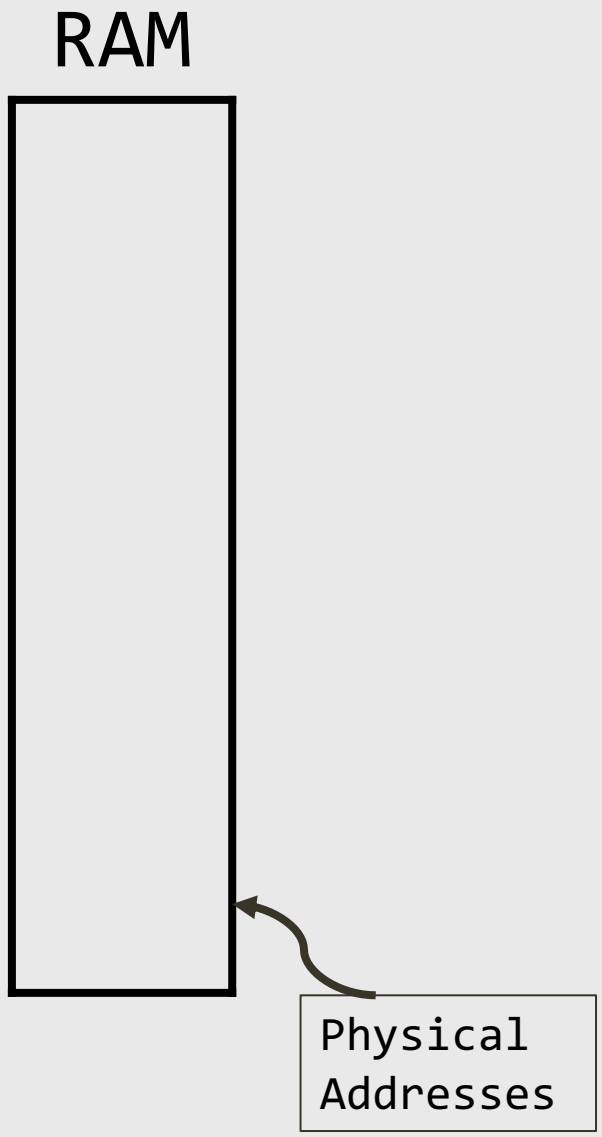
0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
0x28	0x29	0x2A	0x2B	0x2C	0x2D	0x2E	0x2F
0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
0x38	0x39	0x3A	0x3B	0x3C	0x3D	0x3E	0x3F
0x40	0x41	0x42	0x43	0x44	0x45	0x46	0x47
0x48	0x49	0x4A	0x4B	0x4C	0x4D	0x4E	0x4F

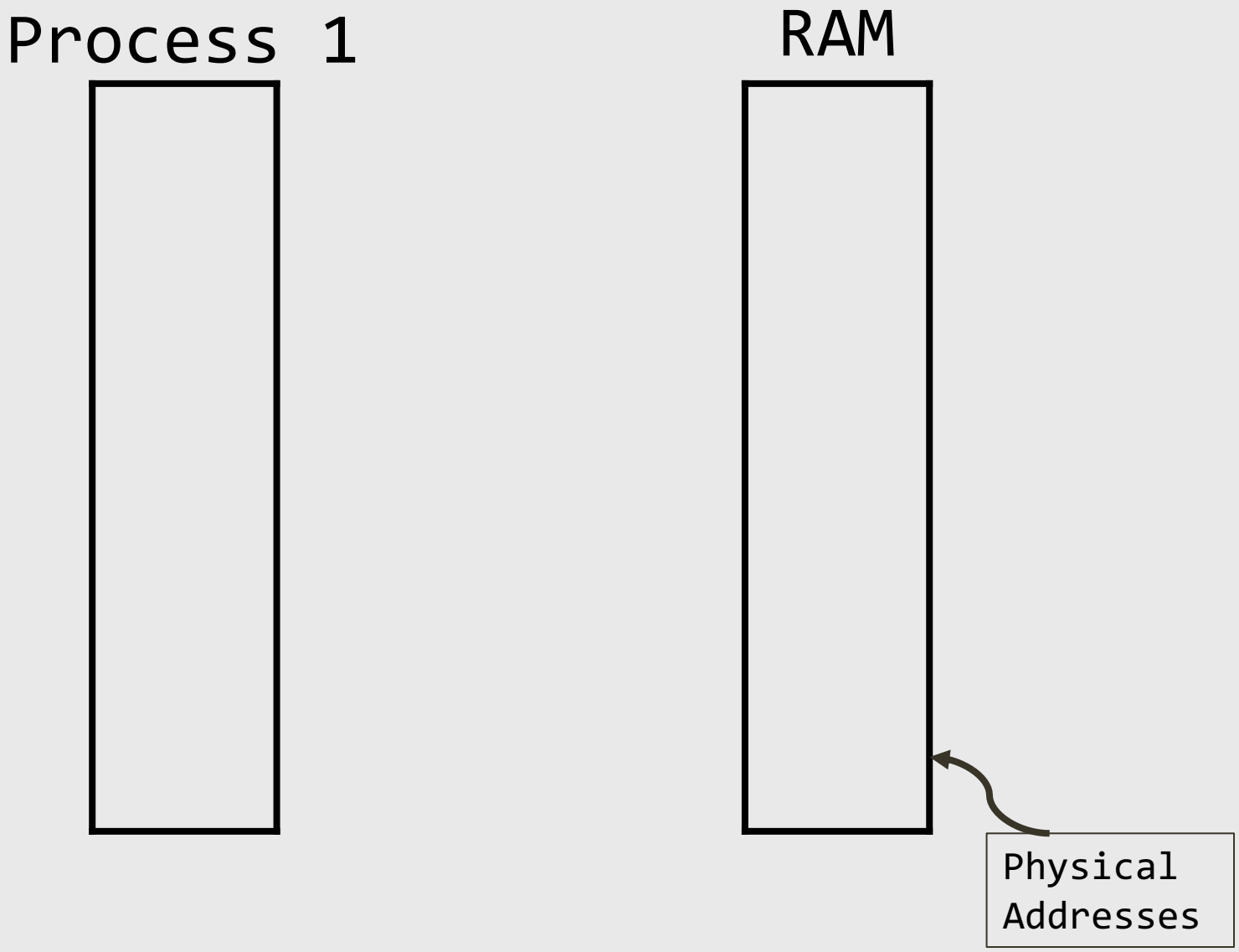
High Memory Addresses

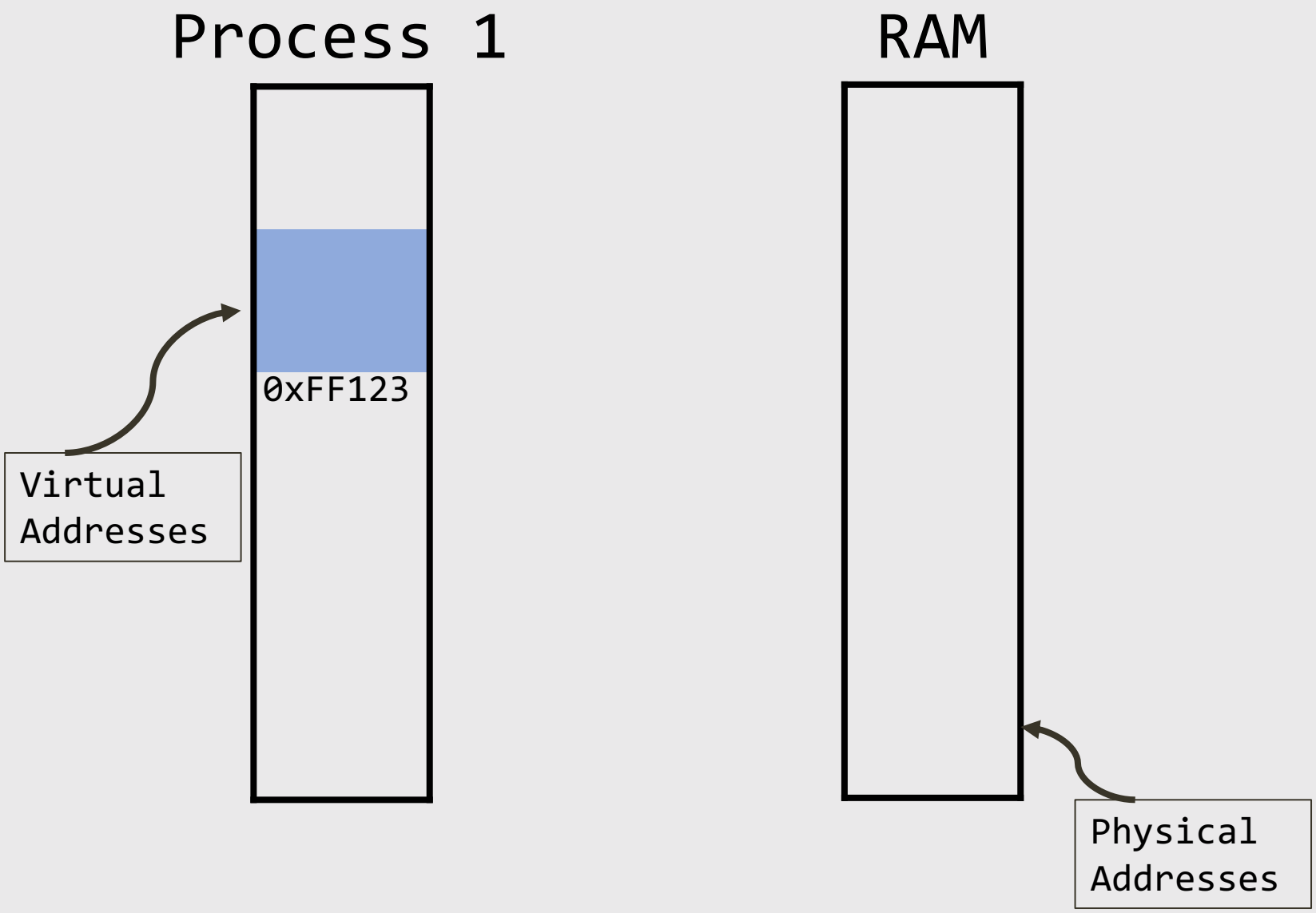

Operating System
User App1
User App2
Your C++ Program
User App3
All of main memory

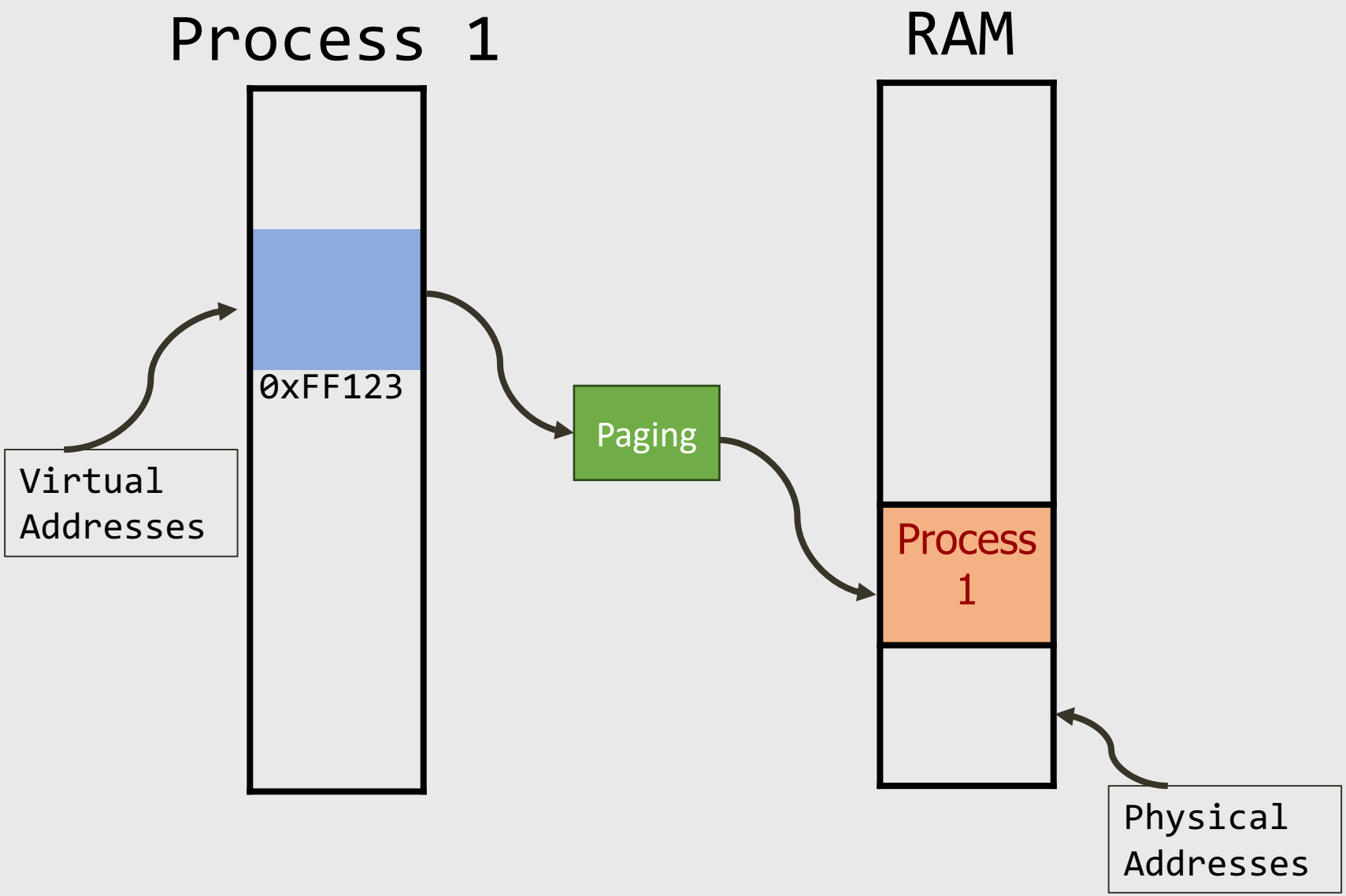

All of main memory
Operating System
User App1
User App2
Your C++ Program
User App3
0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7
0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17
0x18 0x19 0x1A 0x1B 0x1C 0x1D 0x1E 0x1F
0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27
0x28 0x29 0x2A 0x2B 0x2C 0x2D 0x2E 0x2F
0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37
0x38 0x39 0x3A 0x3B 0x3C 0x3D 0x3E 0x3F
0x40 0x41 0x42 0x43 0x44 0x45 0x46 0x47
0x48 0x49 0x4A 0x4B 0x4C 0x4D 0x4E 0x4F

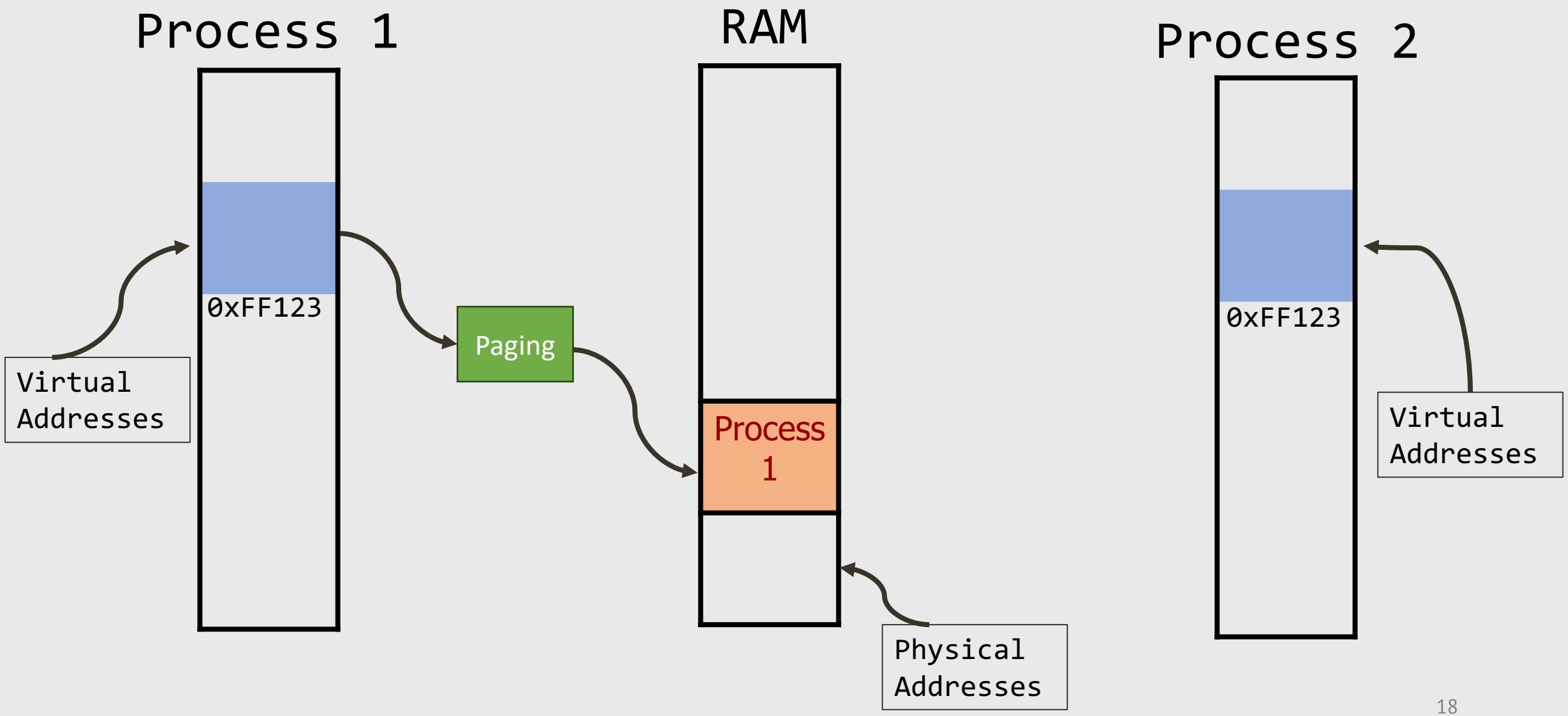
Virtual vs. Physical Addresses

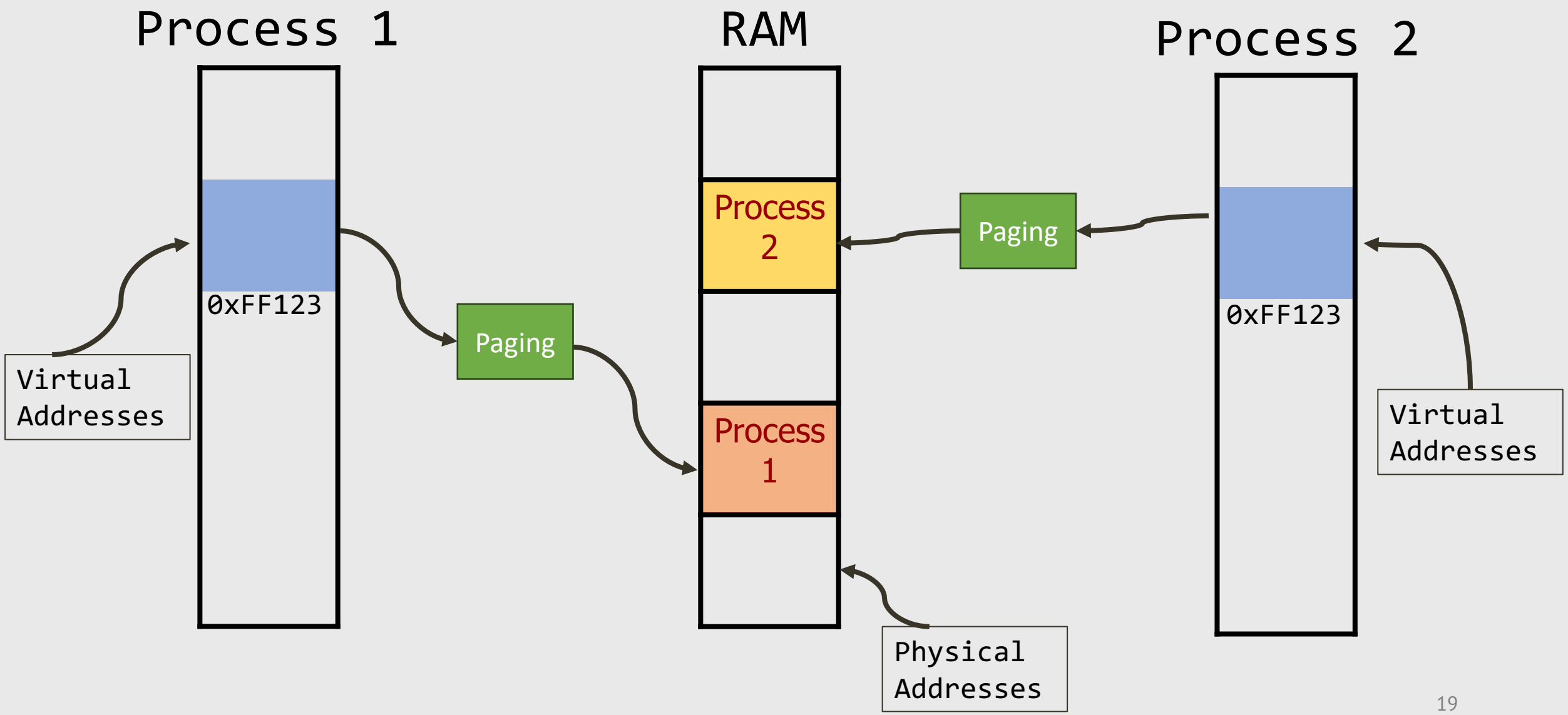



Process 1
RAM
Physical
Addresses

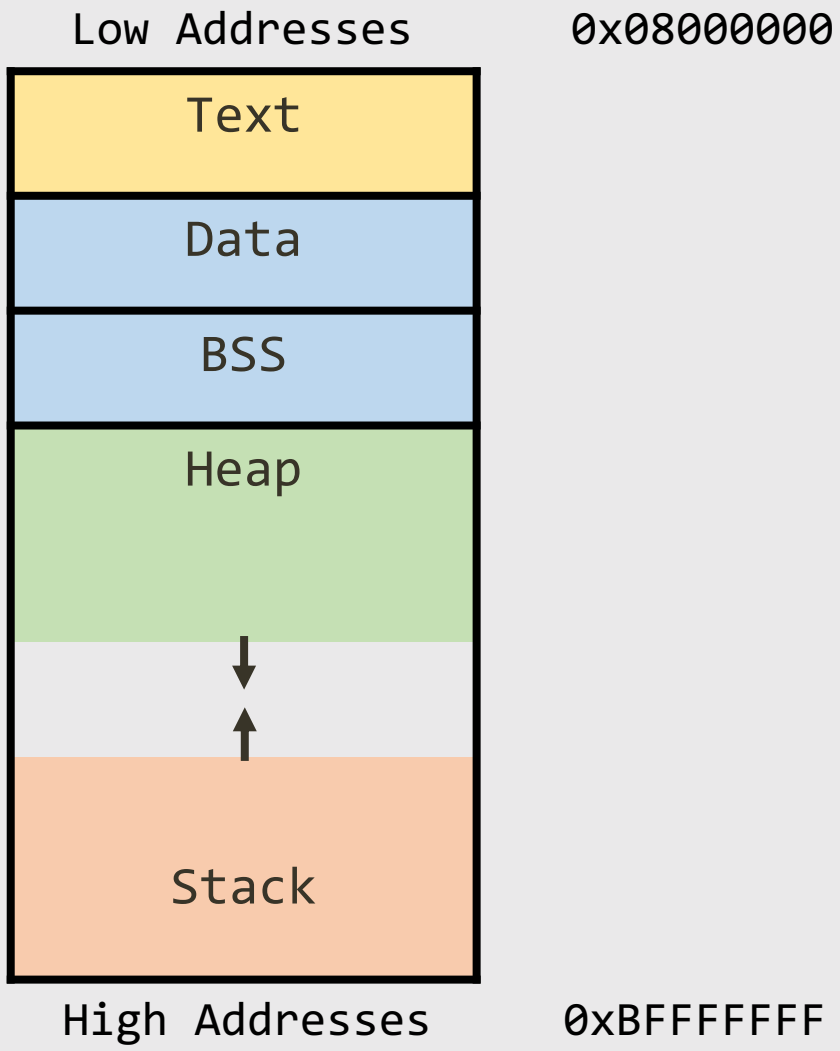

Process 1
RAM
0xFF123
Virtual Addresses
Physical Addresses


Process 1
RAM
0xFF123
Virtual Addresses
Paging
Process 1
Physical Addresses

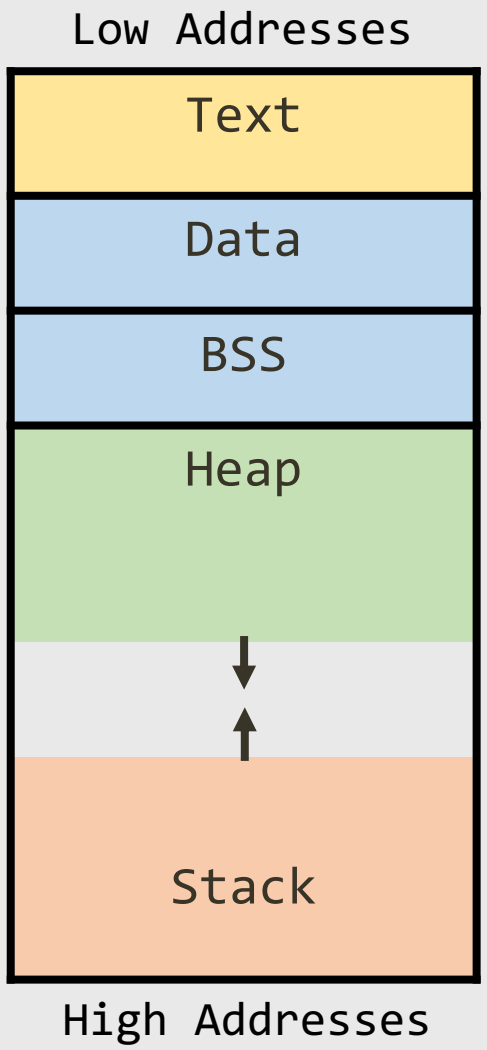

Process 1
RAM
Process 2
0xFF123
Paging
Virtual Addresses
Process 1
0xFF123
Virtual Addresses
Physical Addresses


Process 1
RAM
Process 2
0xFF123
Paging
Process 2
Paging
0xFF123
Virtual Addresses
Process 1
Virtual Addresses
Physical Addresses

Memory Layout



Memory Layout



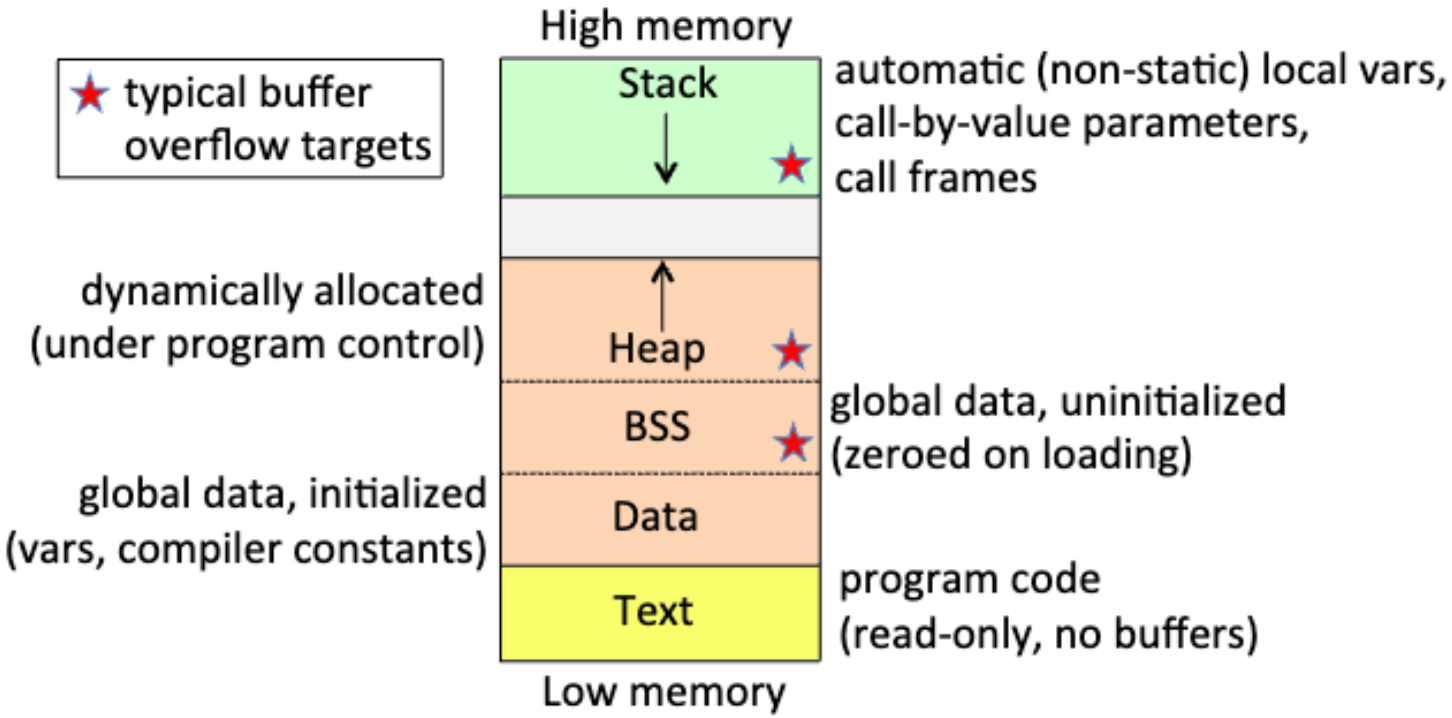
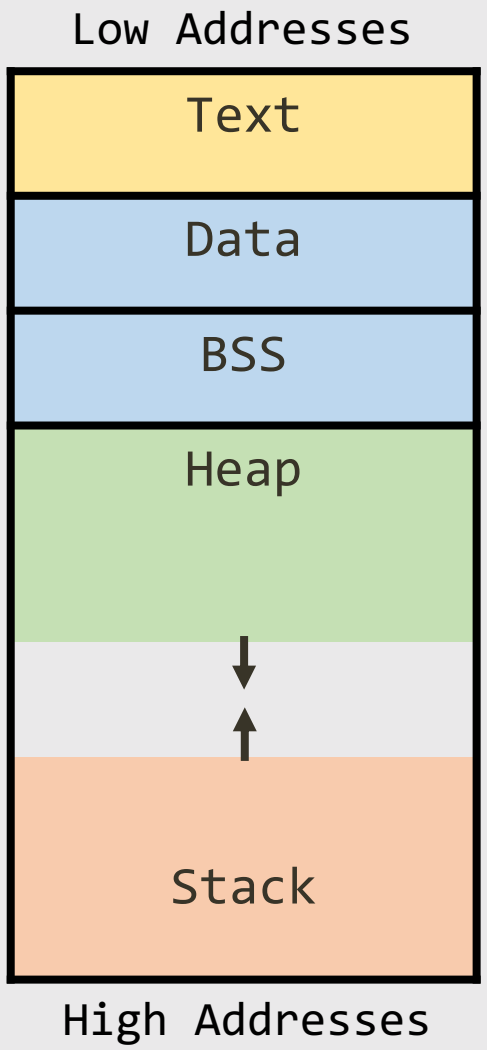
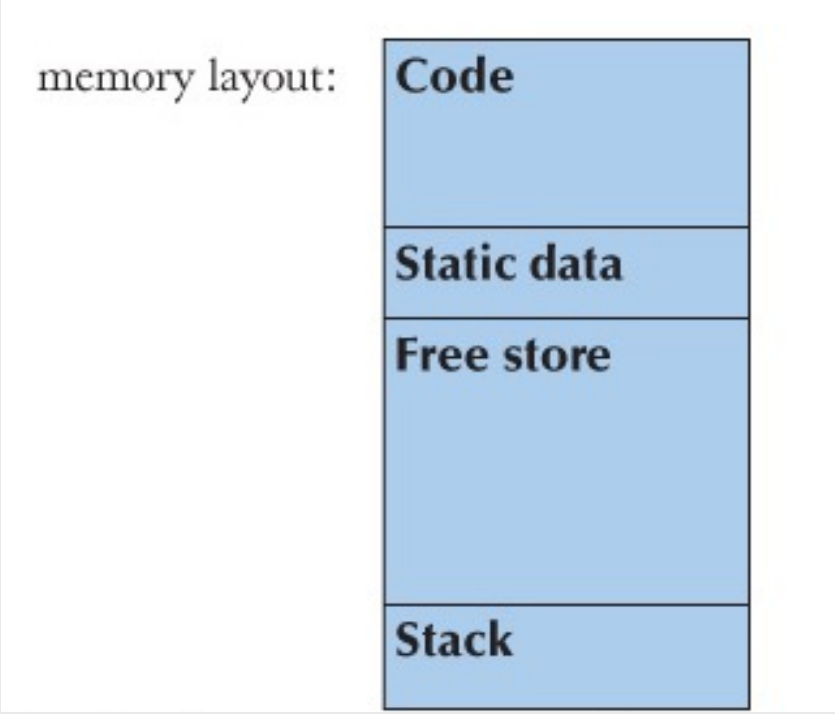


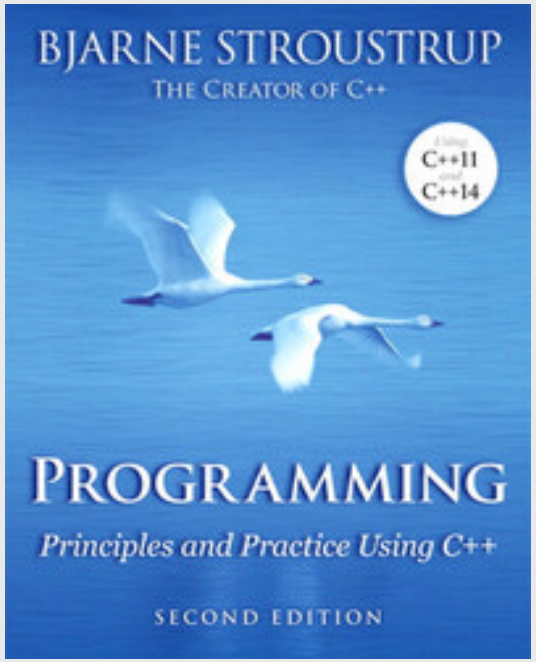
Figure 6.3: Common memory layout (user-space processes).

https://people.scs.carleton.ca/~paulv/toolsjewels/TJrev1/ch6-rev1.pdf

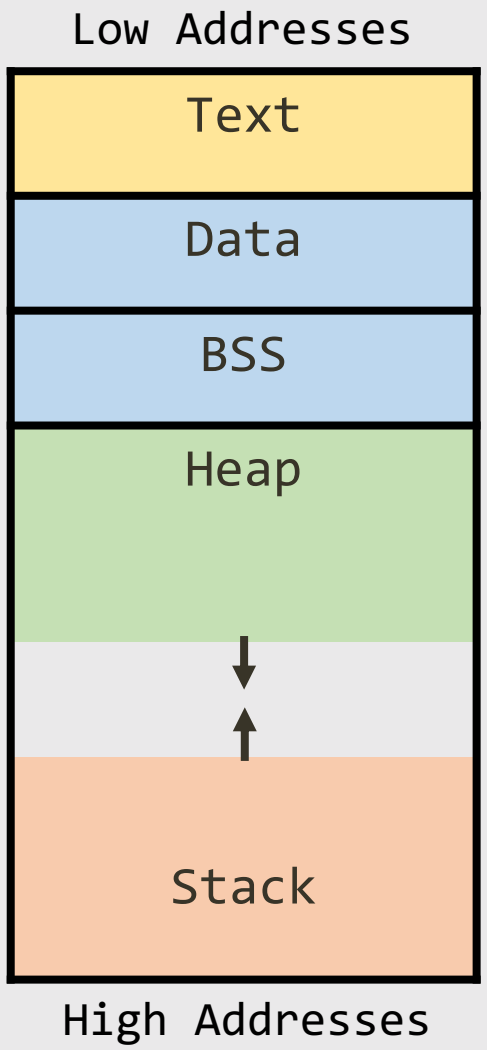
Memory Layout



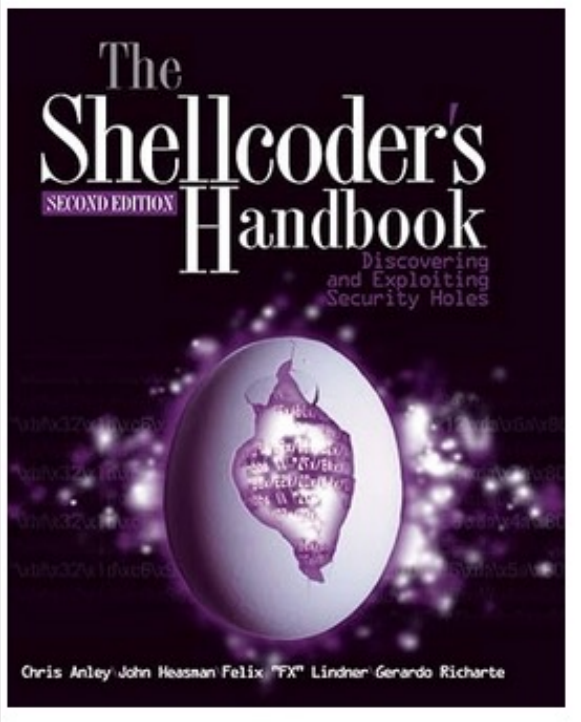




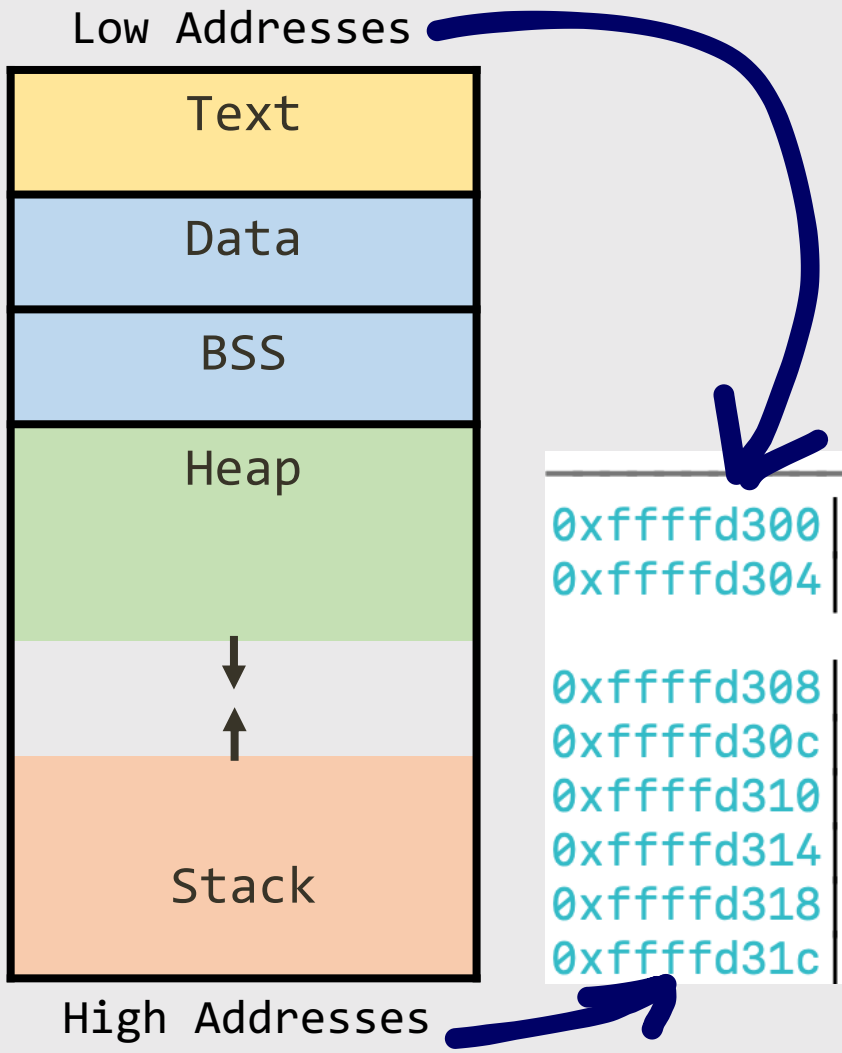
Memory Layout



↑ Lower addresses (0x08000000)
Shared libraries
.text
.bss
Heap (grows ↓)
Stack (grows ↑)
env pointer
Argc
↓ Higher addresses (0xbfffffff)



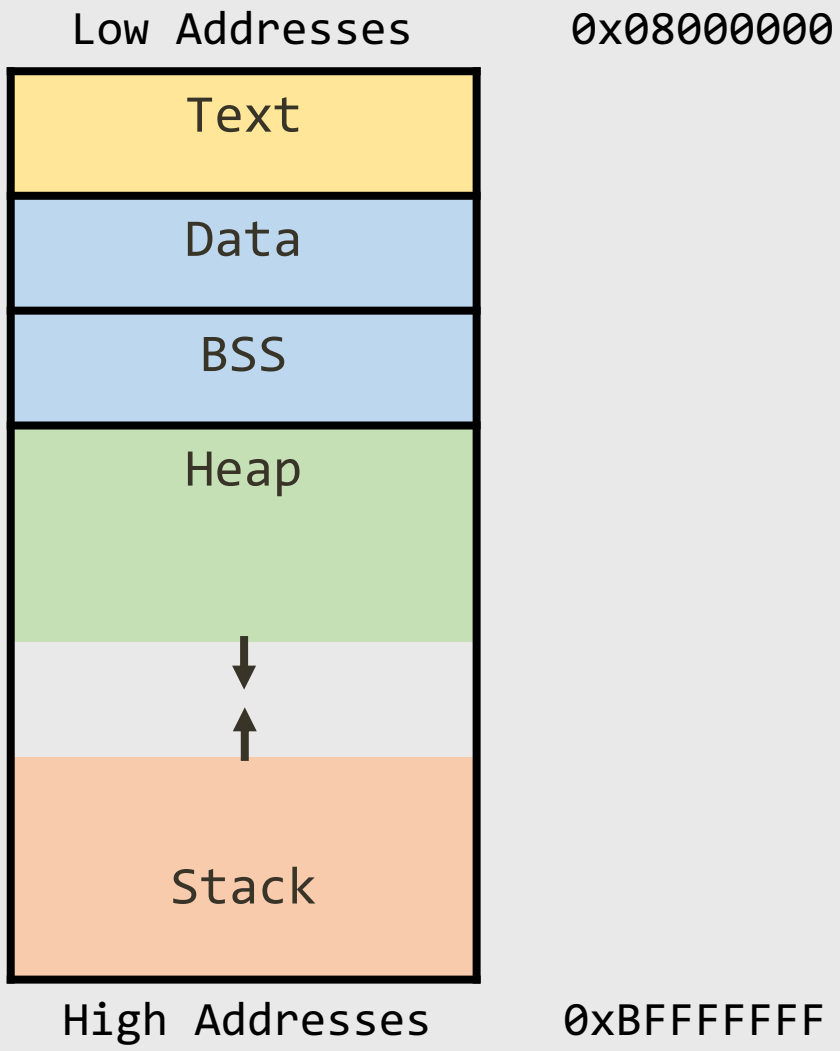
Memory Layout

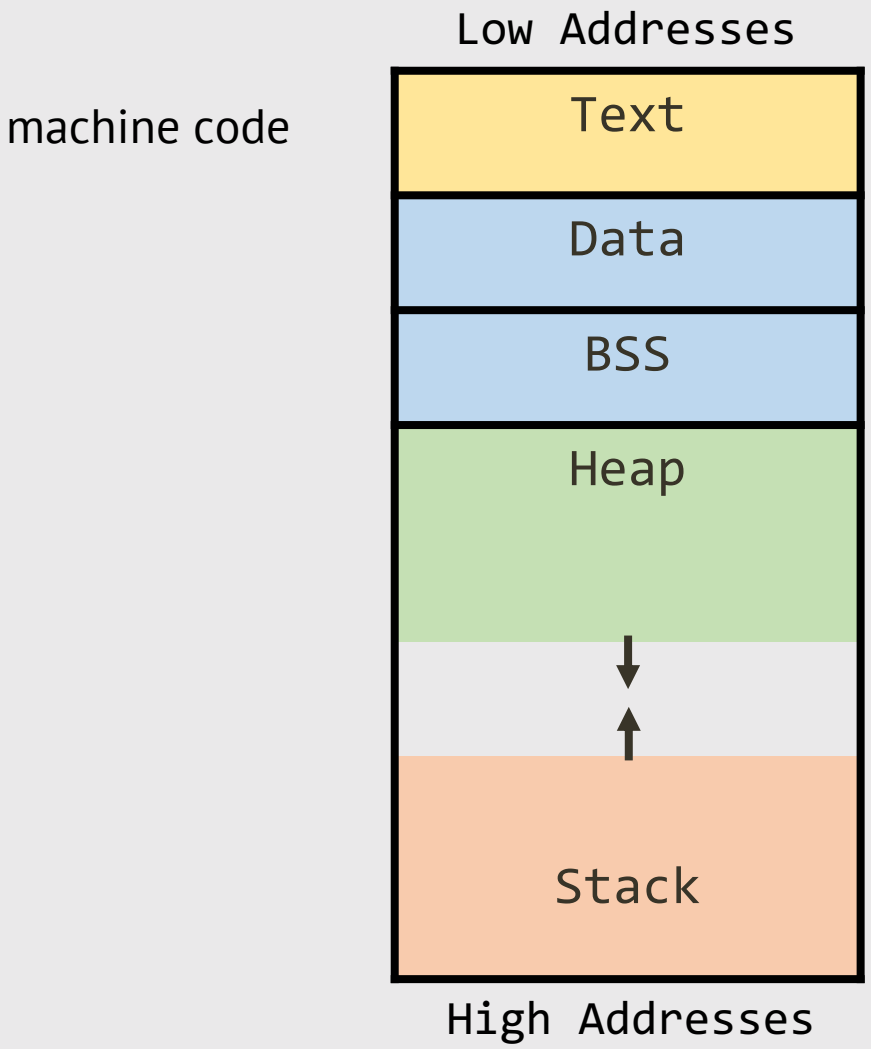


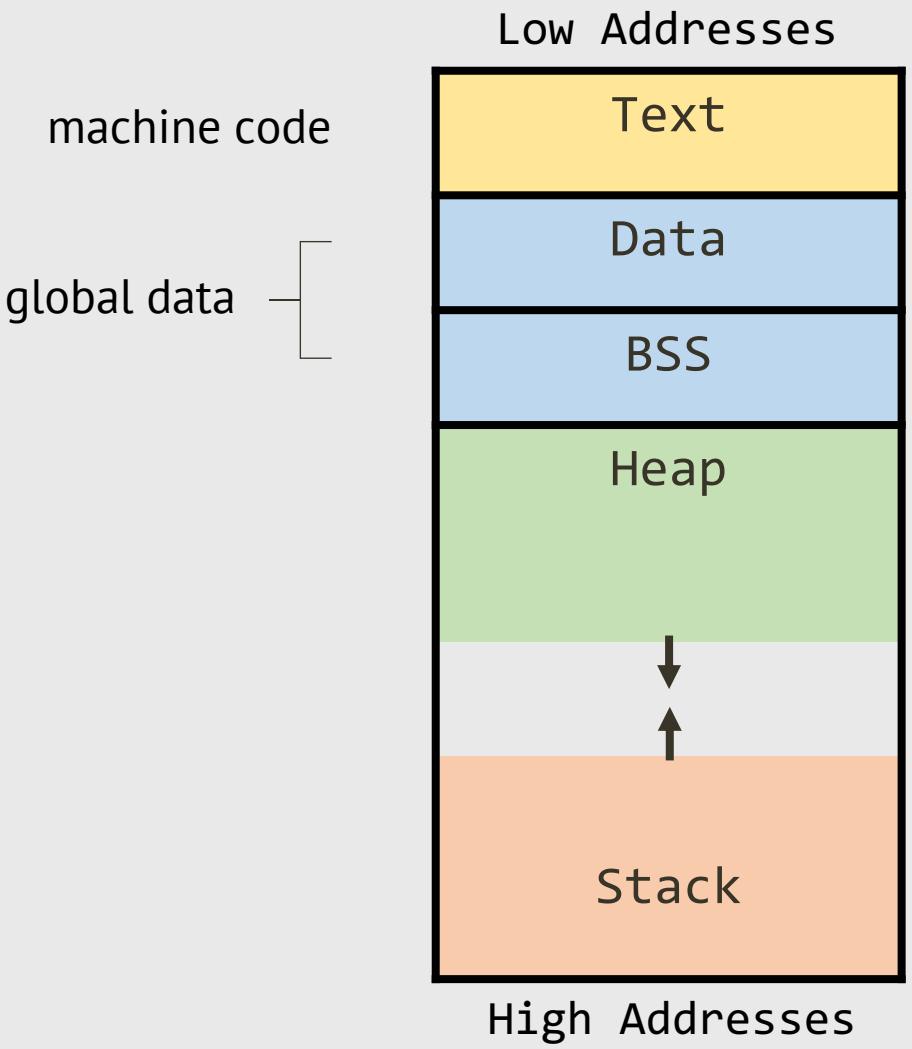
GNU Debugger (GDB)

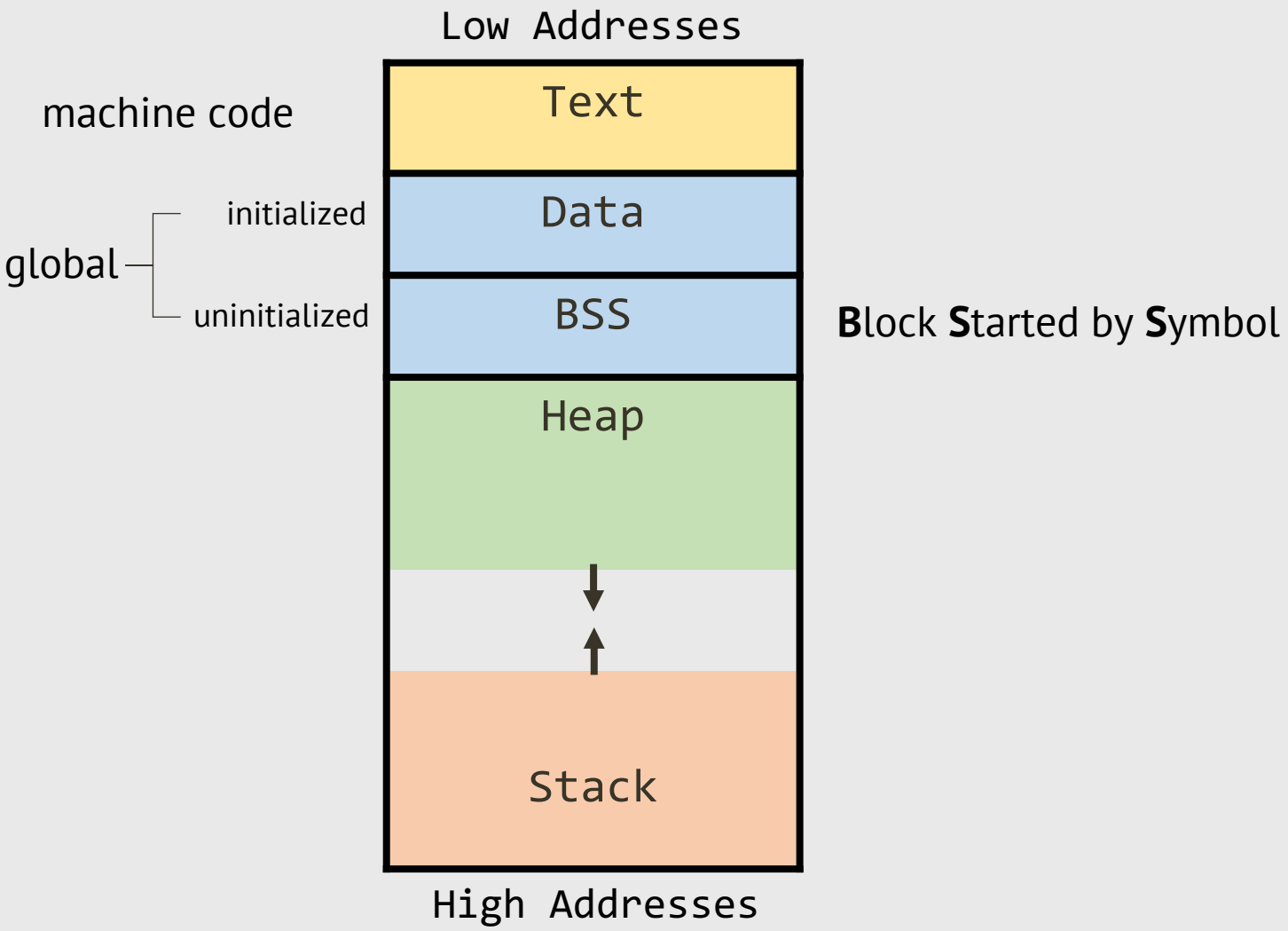
```
──────────────────────────────────────────────────────────────────────────────── stack
0xffffd300│+0x0000: 0xf7ffcff4  →  0x00032f34    ← $esp
0xffffd304│+0x0004: 0x0000000c ("
                                  "?)
0xffffd308│+0x0008: 0x00000000
0xffffd30c│+0x000c: 0x00000000
0xffffd310│+0x0010: 0xffffd338  →  0x00000000    ← $ebp
0xffffd314│+0x0014: 0x565561e3  →  <main+54> add esp, 0x8
0xffffd318│+0x0018: 0x0000000a ("\n"?)
0xffffd31c│+0x001c: 0x00000014
```

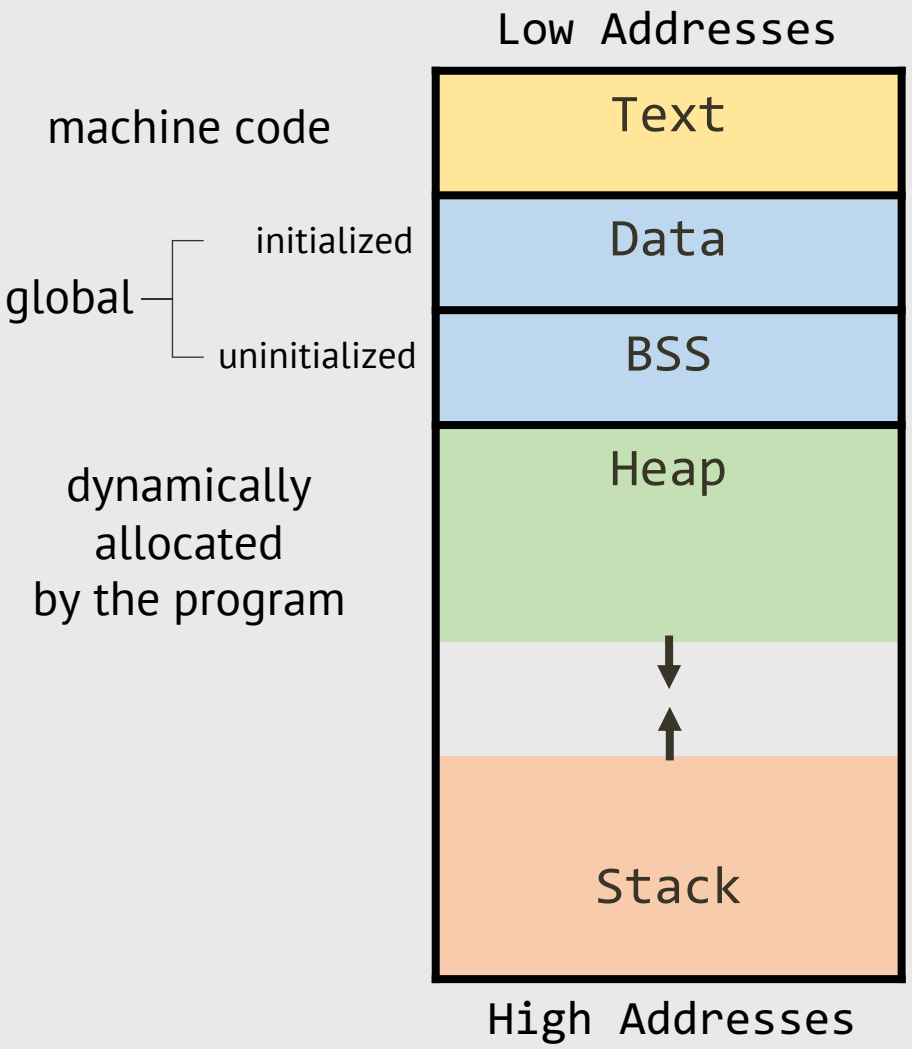
Memory Layout

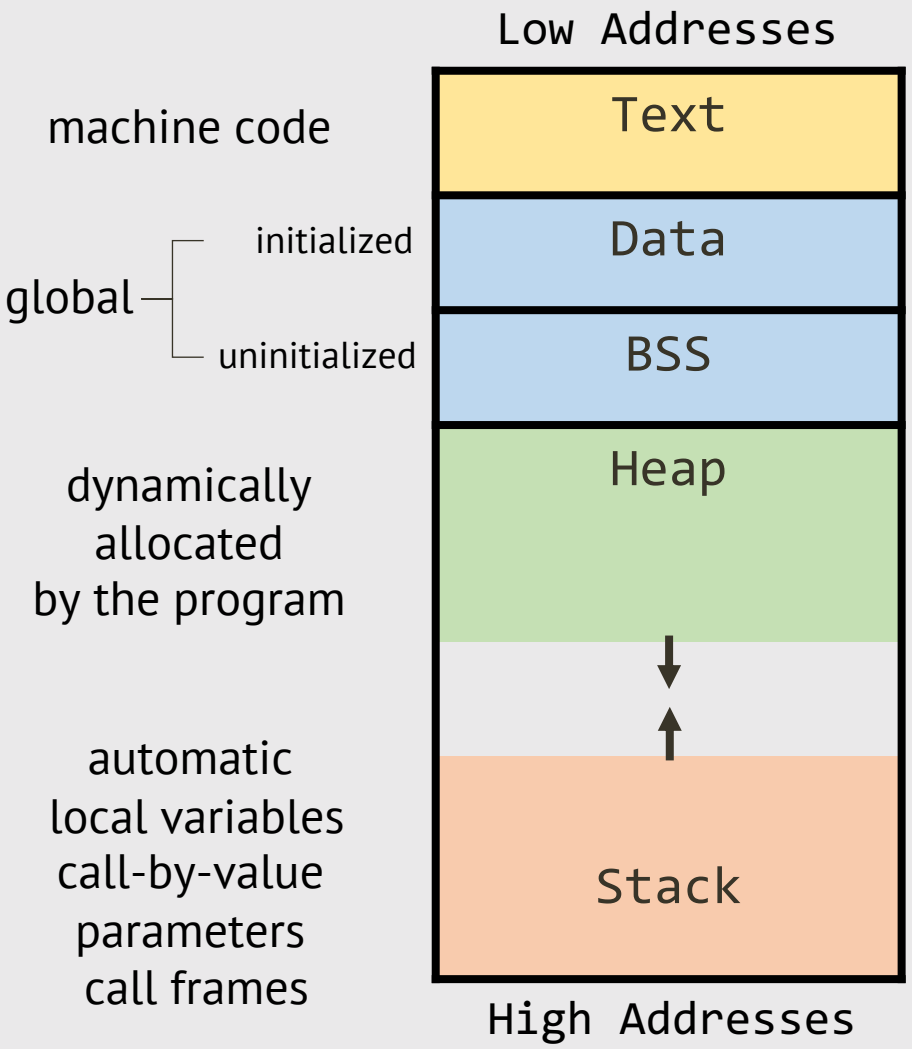


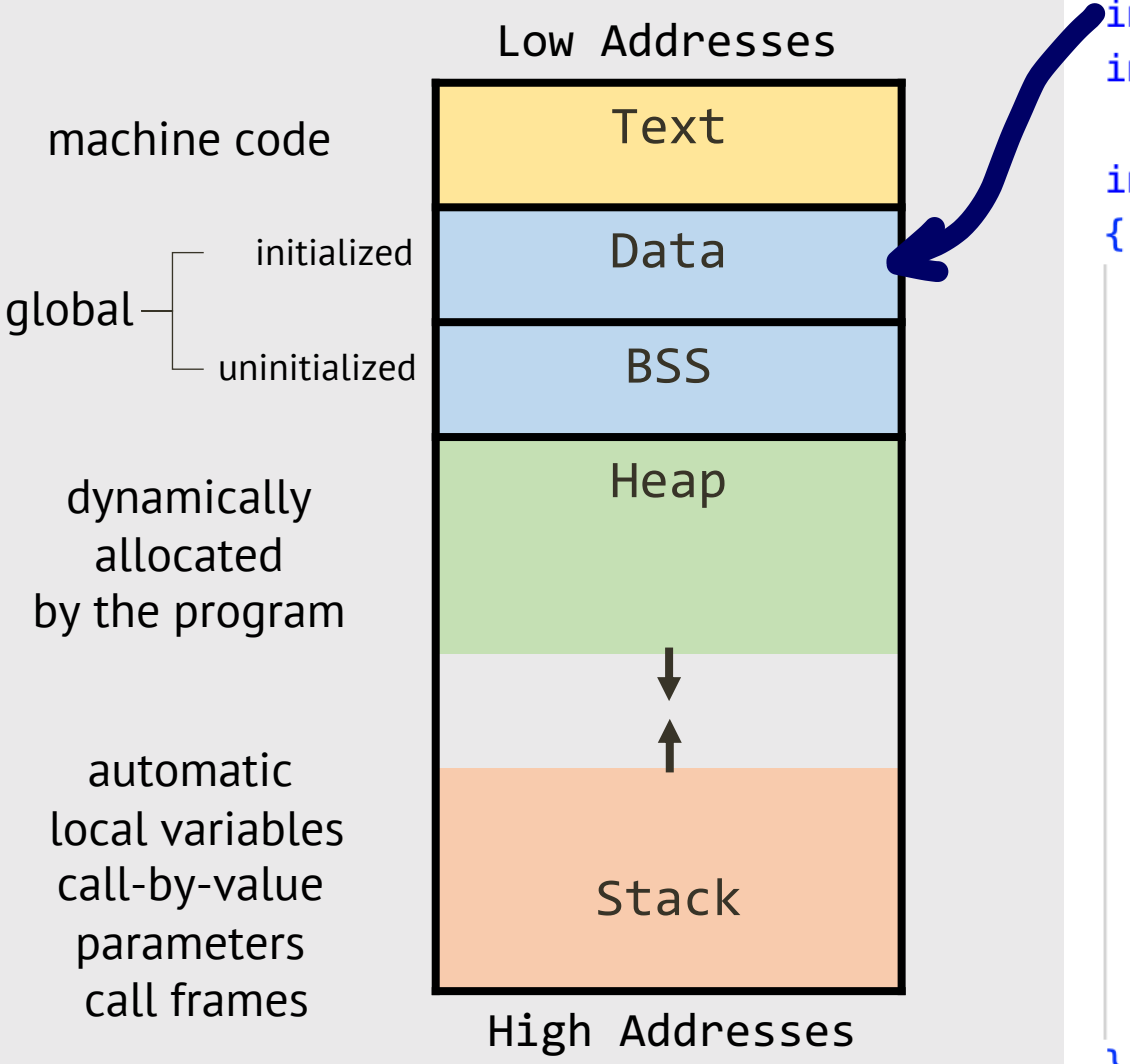

Low Addresses
machine code
Text
Data
BSS
Heap
Stack
High Addresses


Low Addresses
machine code
Text
Data
global data
BSS
Heap
Stack
High Addresses

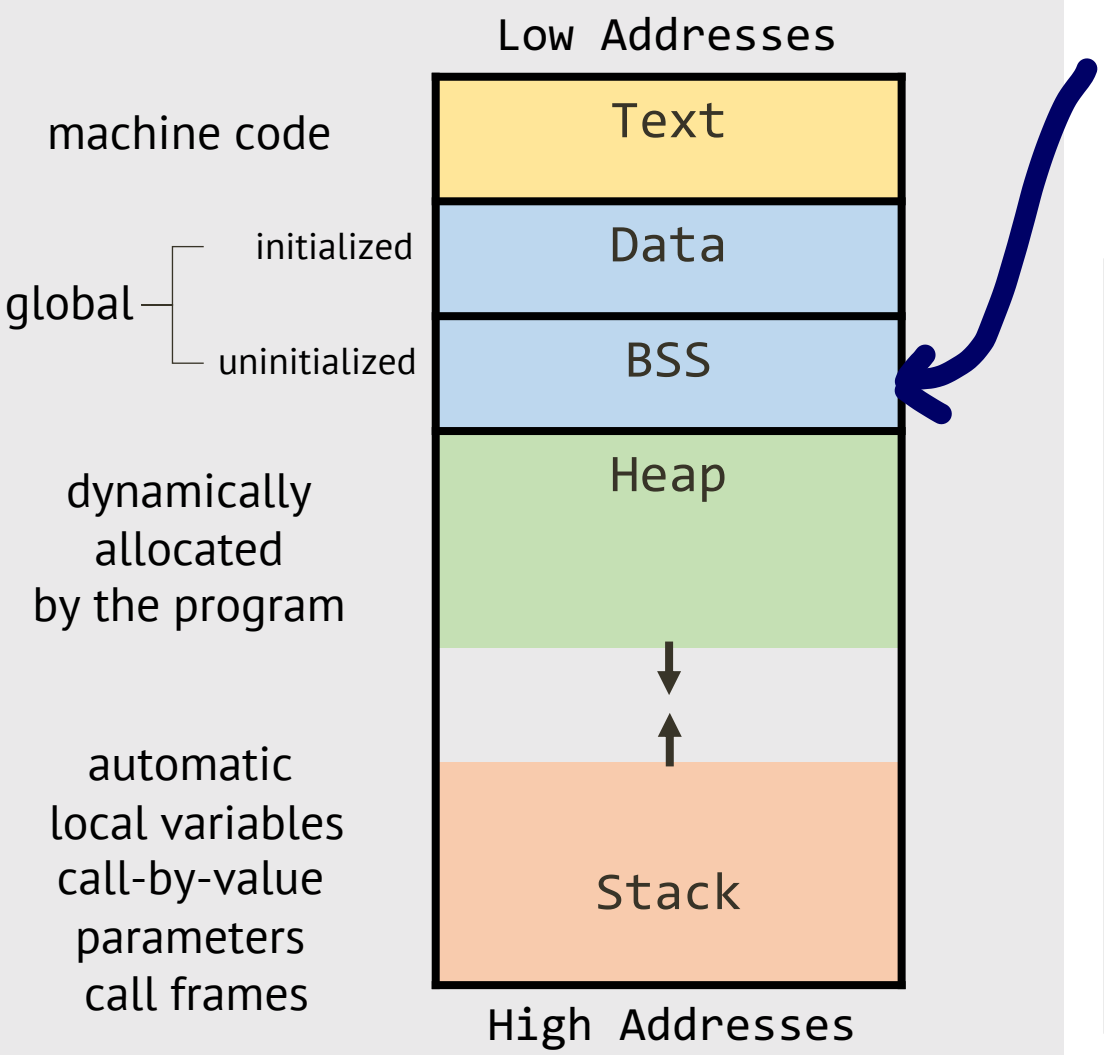

Low Addresses
machine code
Text
initialized
Data
global
uninitialized
BSS
Block Started by Symbol
Heap
Stack
High Addresses


Low Addresses
machine code
Text
initialized
Data
global
uninitialized
BSS
Heap
dynamically
allocated
by the program
Stack
High Addresses

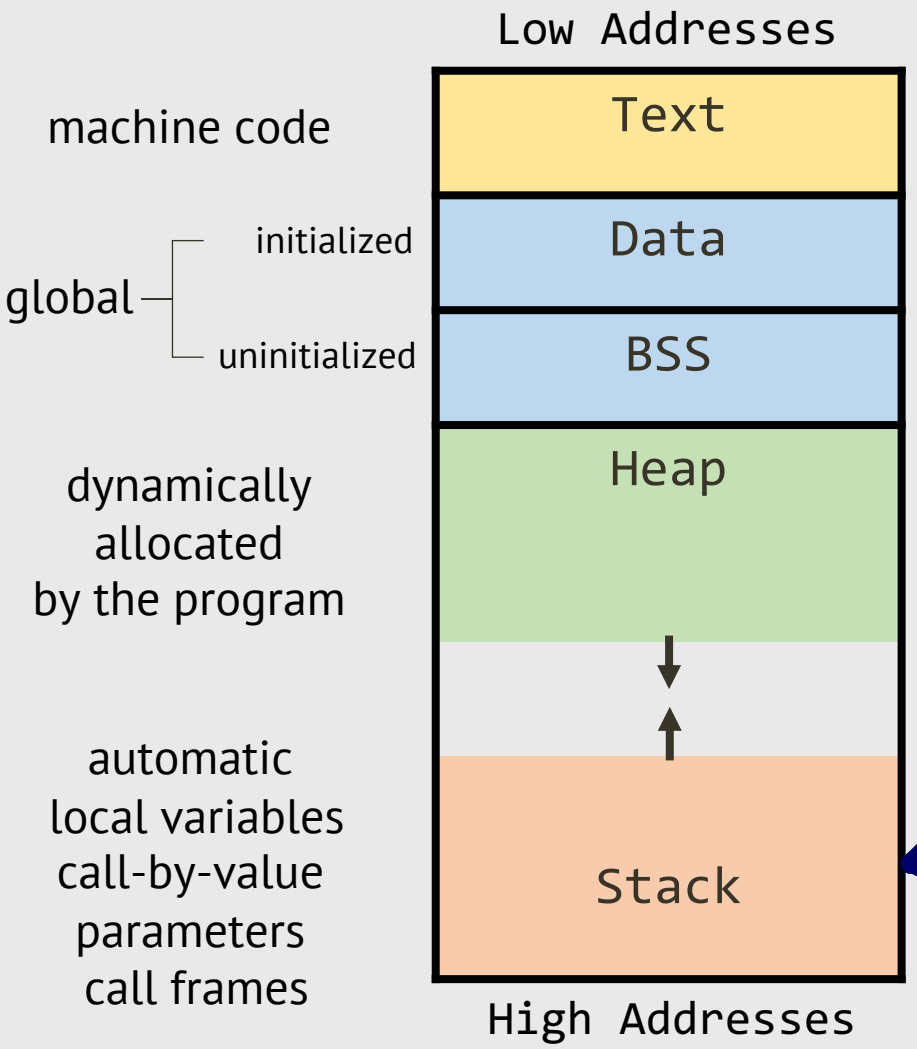

Low Addresses
machine code
Text
initialized
Data
global
uninitialized
BSS
Heap
dynamically
allocated
by the program
automatic
local variables
call-by-value
parameters
call frames
Stack
High Addresses



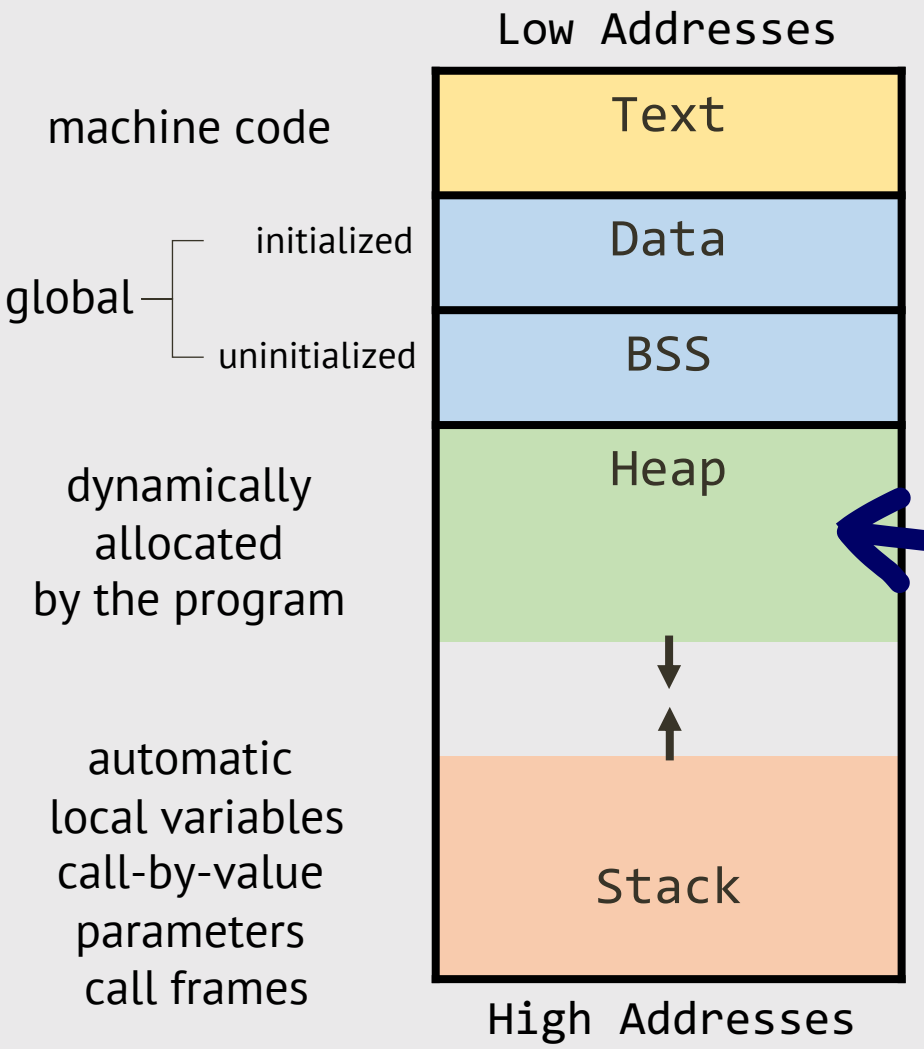

```cpp
int x = 100;    // Initialized Global Varaible.
int y;          // Uninitialized

int main()
{
    // Data stored on the stack in the main stack frame.
    int a = 2;
    float b = 2.5;

    // Allocate memory on the heap.  malloc(2 * sizeof(int))
    int* ptr = new int[2];

    // Values store on heap.
    ptr[0] = 5;
    ptr[1] = 6;

    delete[] ptr;  // free(ptr)

    return 1;
}
```



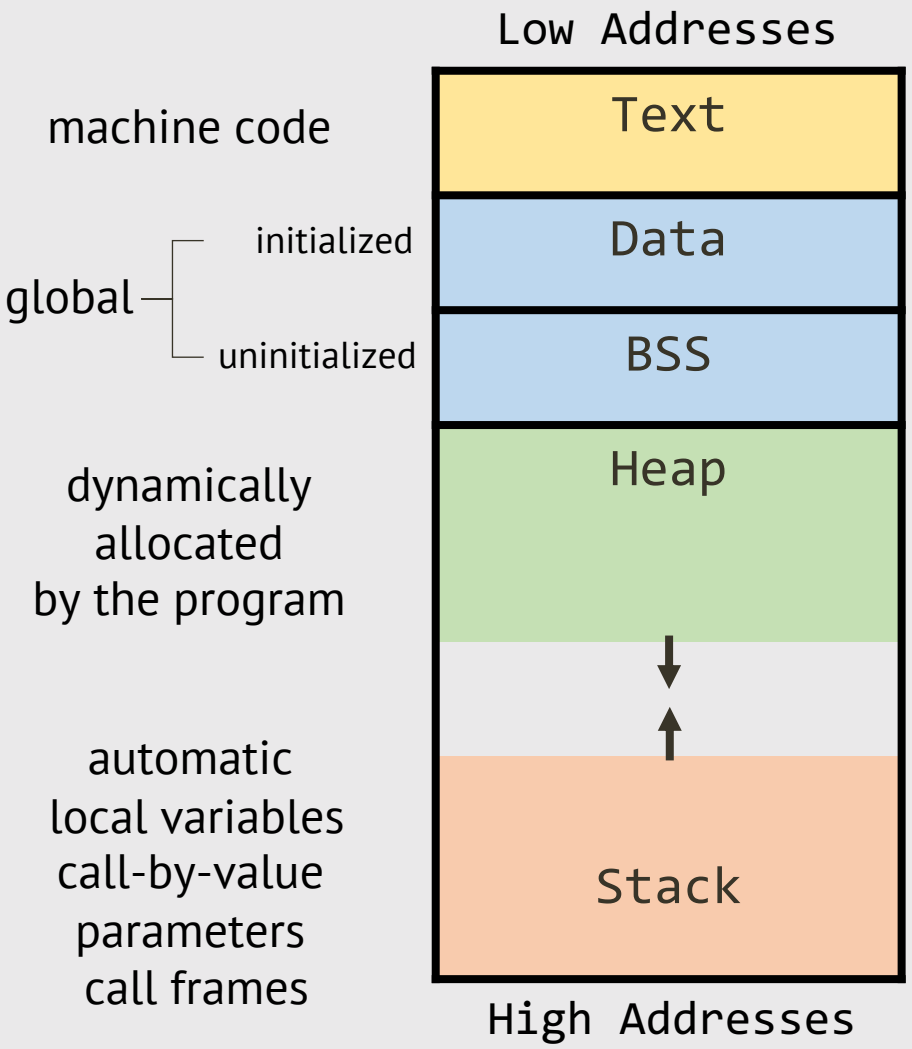

```
int x = 100;     // Initialized Global Varaible.
int y;           // Uninitialized

int main()
{
    // Data stored on the stack in the main stack frame.
    int a = 2;
    float b = 2.5;

    // Allocate memory on the heap.  malloc(2 * sizeof(int))
    int* ptr = new int[2];

    // Values store on heap.
    ptr[0] = 5;
    ptr[1] = 6;

    delete[] ptr;  // free(ptr)

    return 1;
}
```




```
int x = 100;     // Initialized Global Varaible.
int y;           // Uninitialized

int main()
{
    // Data stored on the stack in the main stack frame.
    int a = 2;
    float b = 2.5;

    // Allocate memory on the heap.  malloc(2 * sizeof(int))
    int* ptr = new int[2];

    // Values store on heap.
    ptr[0] = 5;
    ptr[1] = 6;

    delete[] ptr;  // free(ptr)

    return 1;
}
```

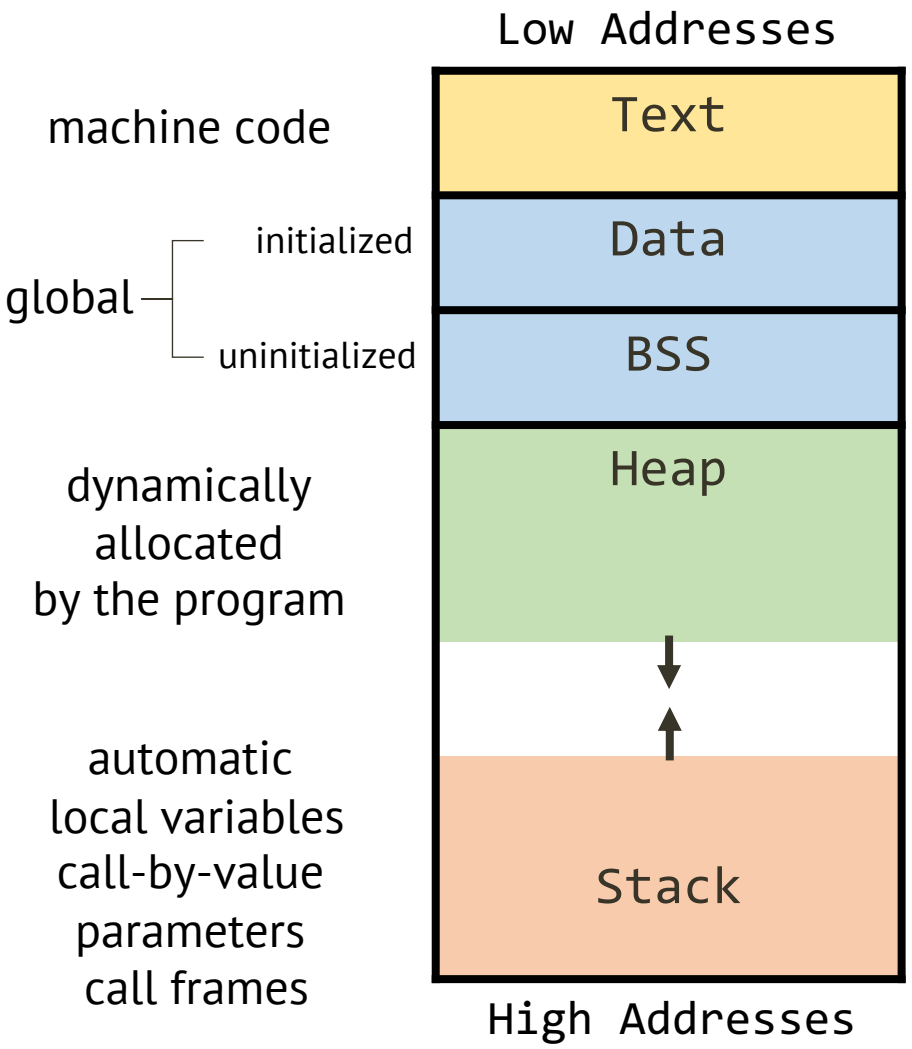



```cpp
int x = 100;    // Initialized Global Varaible.
int y;          // Uninitialized

int main()
{
    // Data stored on the stack in the main stack frame.
    int a = 2;
    float b = 2.5;

    // Allocate memory on the heap.  malloc(2 * sizeof(int))
    int* ptr = new int[2];

    // Values store on heap.
    ptr[0] = 5;
    ptr[1] = 6;

    delete[] ptr;  // free(ptr)

    return 1;
}
```

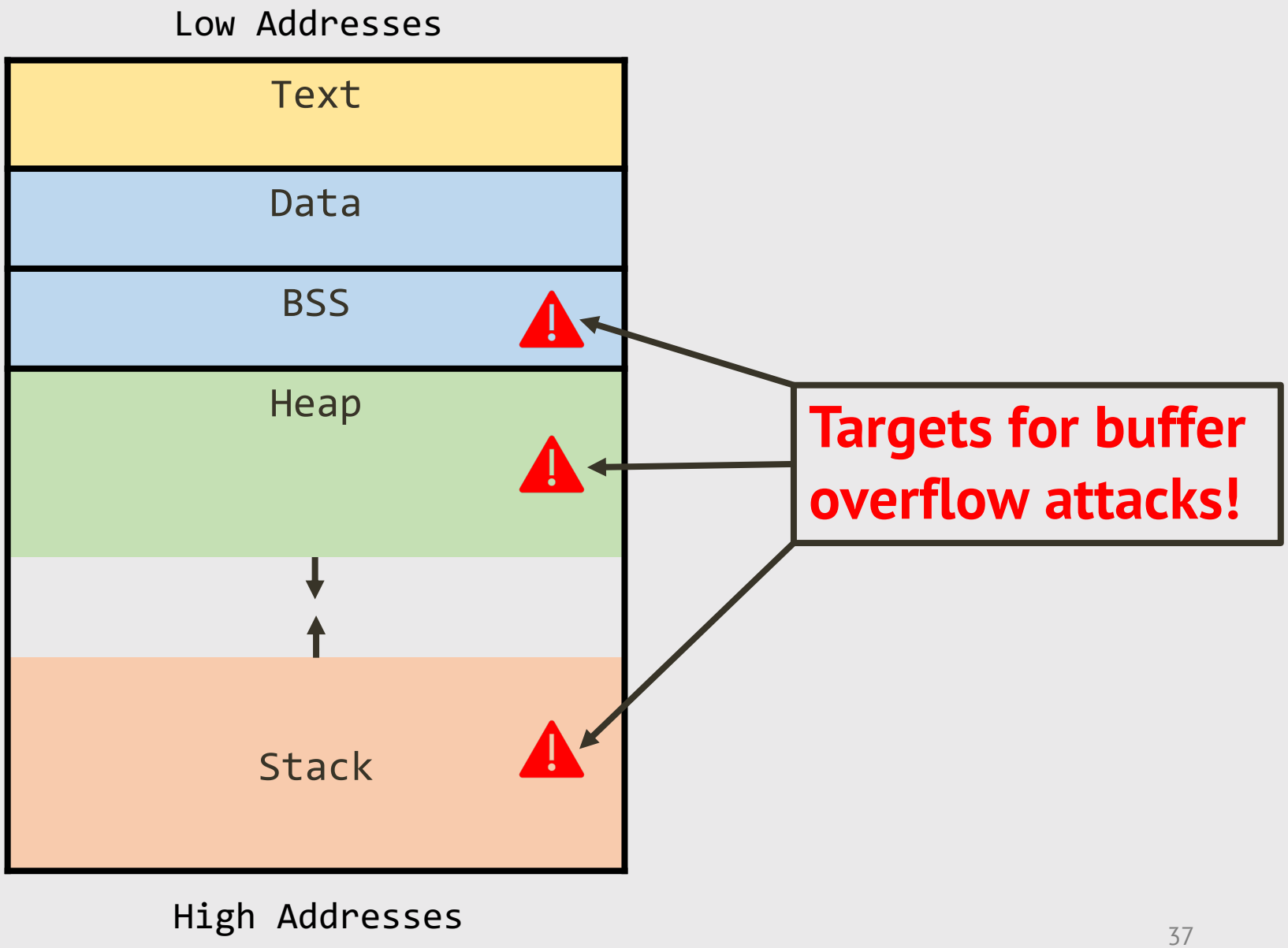


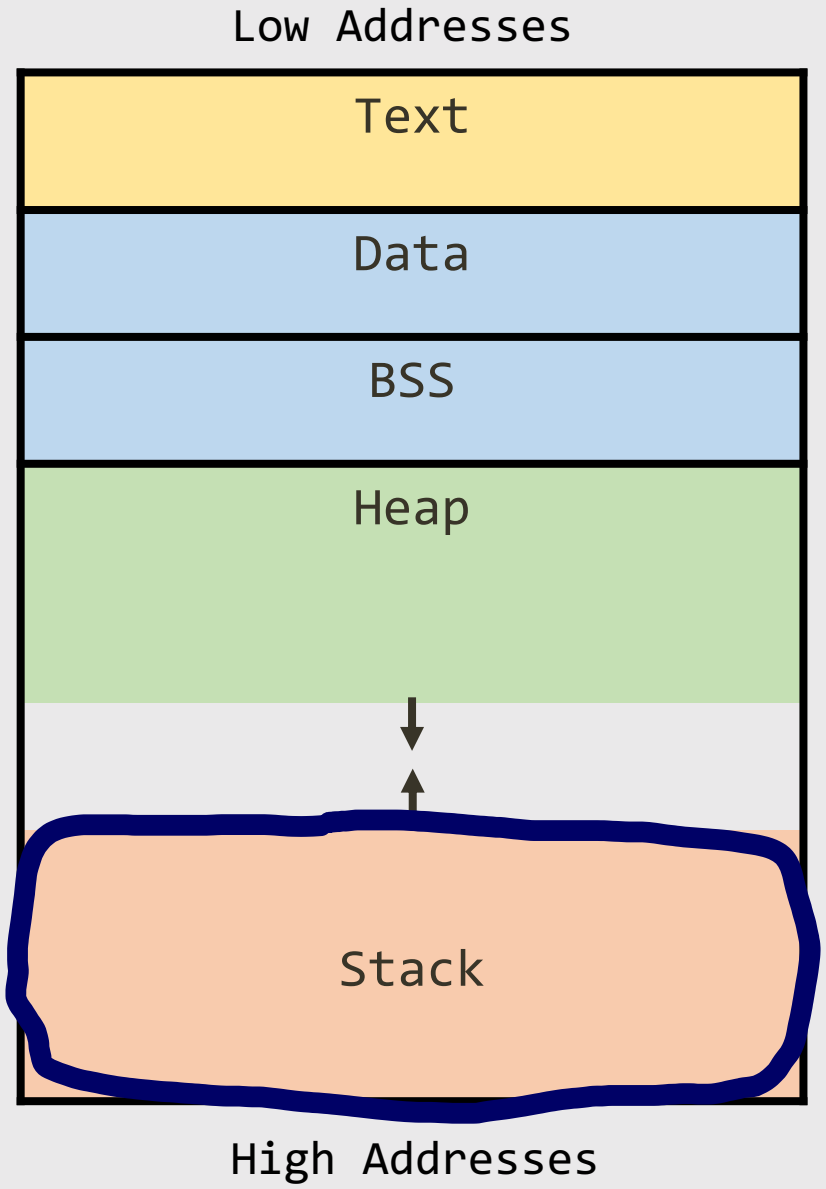

```
int x = 100;     // Initialized Global Varaible.
int y;           // Uninitialized

int main()
{
    // Data stored on the stack in the main stack frame.
    int a = 2;
    float b = 2.5;

    // Allocate memory on the heap.  malloc(2 * sizeof(int))
    int* ptr = new int[2];

    // Values store on heap.
    ptr[0] = 5;
    ptr[1] = 6;

    delete[] ptr;  // free(ptr)

    return 1;
}
```

Draw lines from variables to memory:




```
int totalLogins = 0;
int sessionCount;

void loginUser(string username)
{
    char* sessionToken = new char[30];

    if(sessionToken != NULL)
    {
        cout << username << " logged in" << endl;
        sessionCount++;
        totalLogins++;
    }

    delete[] sessionToken;
}

int main()
{
    string username = "Alice";

    loginUser(username);
    return 0;
}
```

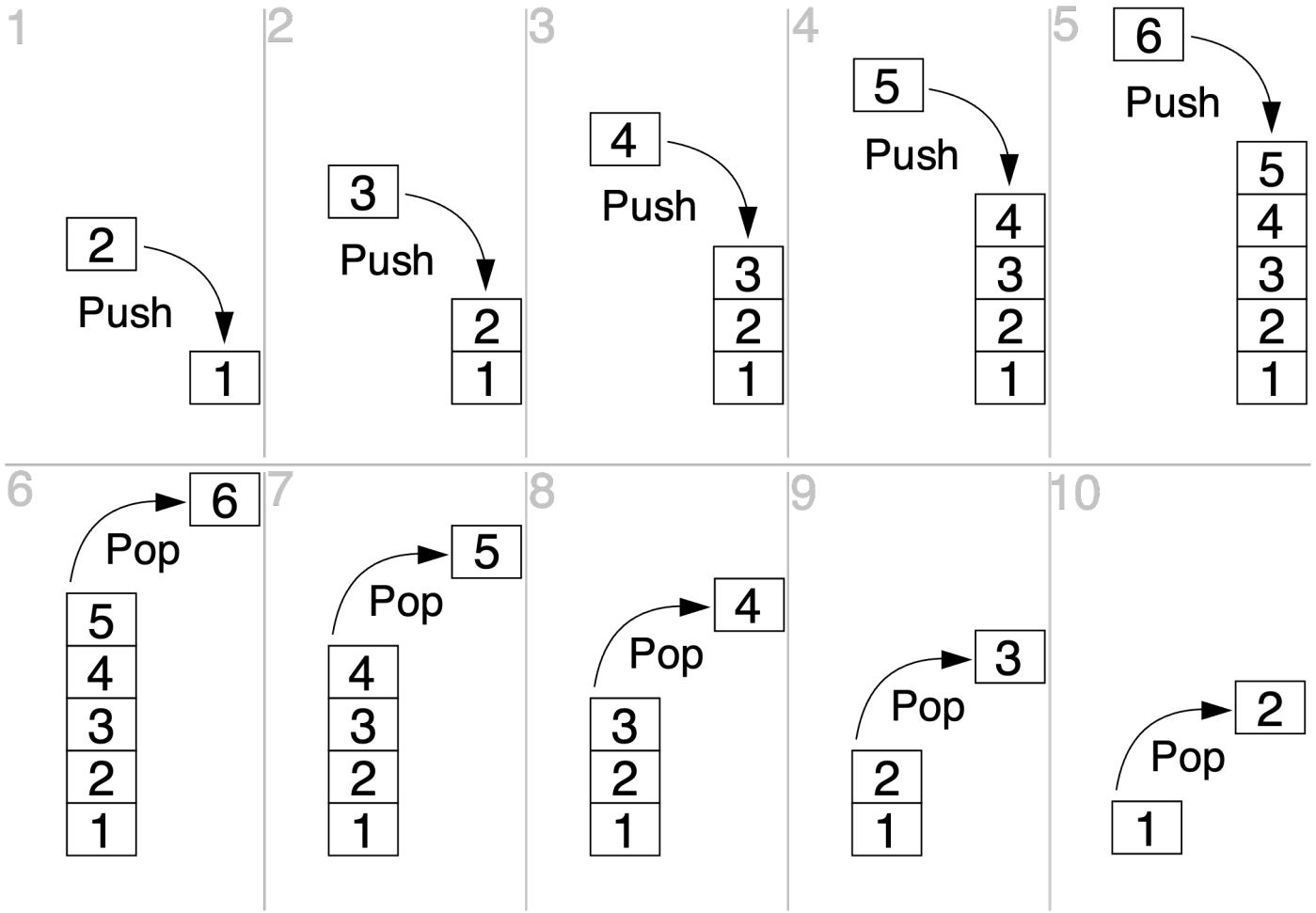

Low Addresses
Text
Data
BSS
Heap
Stack
High Addresses
Targets for buffer overflow attacks!


Low Addresses
Text
Data
BSS
Heap
Stack
High Addresses

Software Vulnerabilities

1. Introduction
2. Program memory layout
3. Stack layout
4. Buffer overflow vulnerability

Stack


1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
6
Pop
5
4
3
2
1
7
5
Pop
4
3
2
1
8
4
Pop
3
2
1
9
3
Pop
2
1
10
2
Pop
1

Call Stack

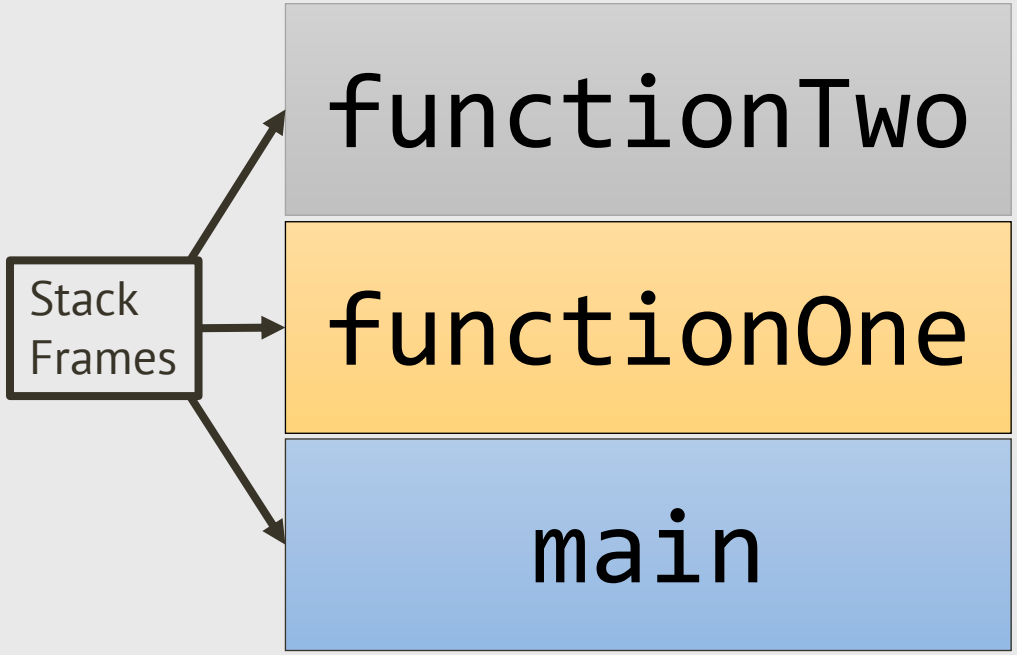
- A **call stack** is a stack data structure that stores information about the active functions of a computer program

- A call stack is composed of **stack frames**

- A stack frame is a data structure used to store information about each function call

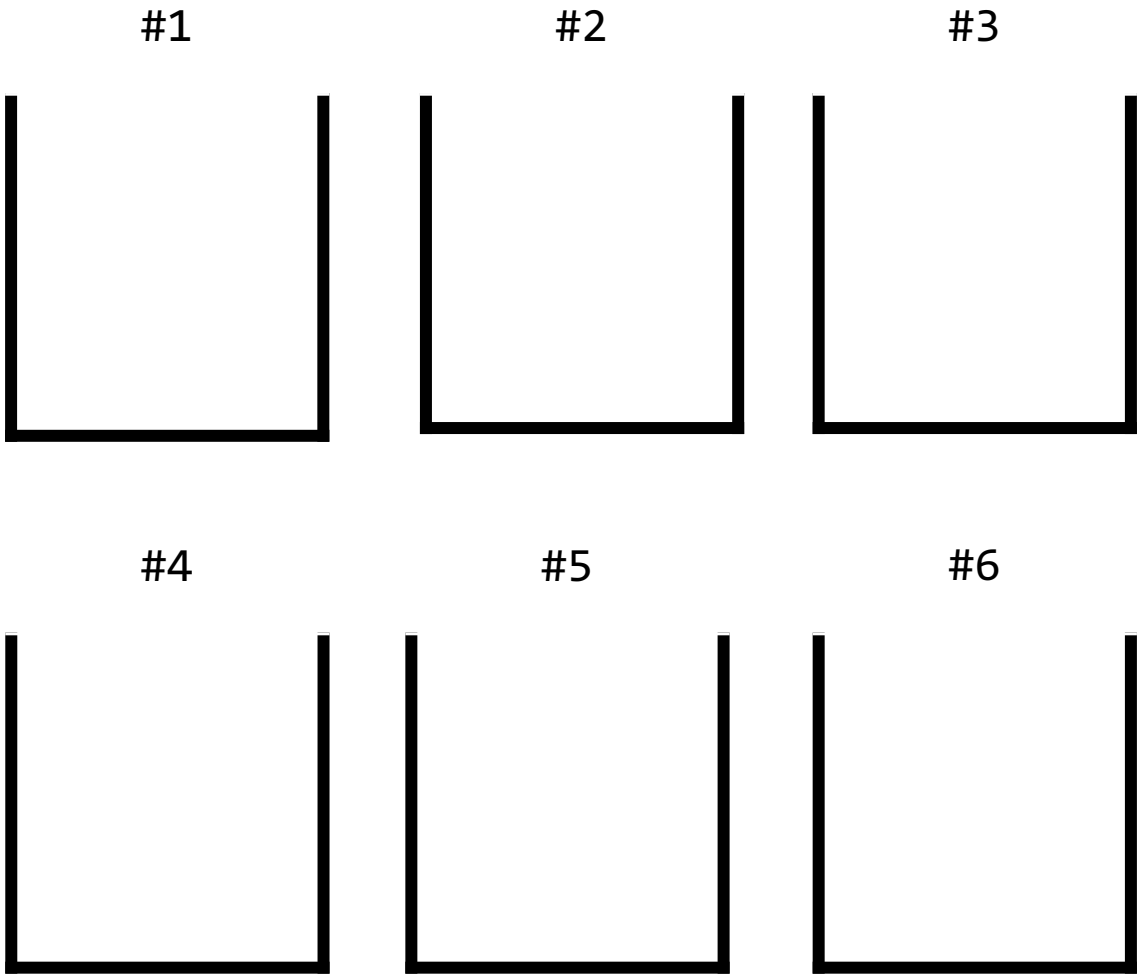
Call Stack

```
void functionTwo()
{
    printf("In function two");
}

void functionOne()
{
    printf("In function one");

    functionTwo();  // Call function Two
}

int main()
{
    functionOne();  // Call function One

    return 0;
}
```



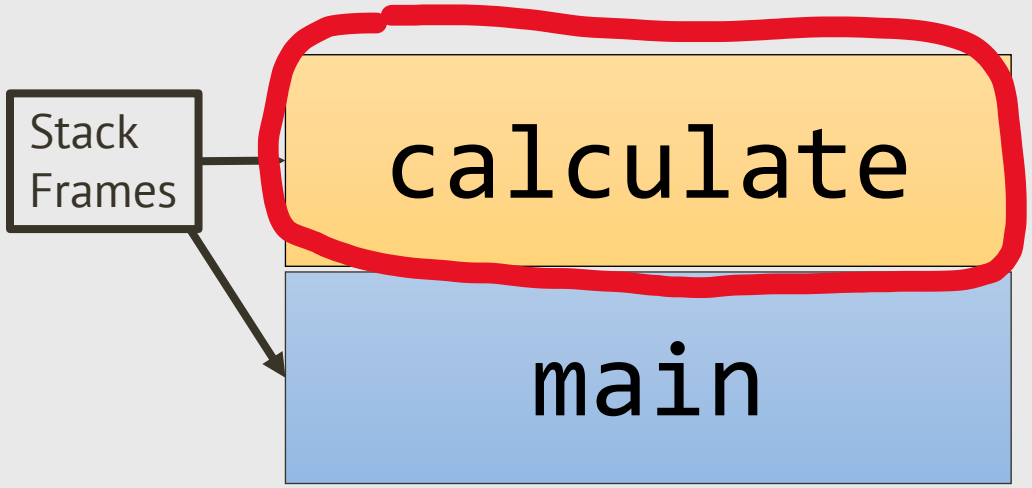
```
int getOne()
{
    int one = 1;                //  CALL STACK #3
    return one;
}

int getTwo()
{
    int two = 2;                //  CALL STACK #4
    return 2;
}

int addOneTwo()
{
    int one = getone();          //  CALL STACK #2
    int two = gettwo();
    return one + two;            //  CALL STACK #5
}

int main()
{
    double pi = 3.14159;          //  CALL STACK #1
    int result = addOneTwo();
    return 0;                     //  CALL STACK #6
}
```

Draw the state of all <u>six</u> **call stacks**:



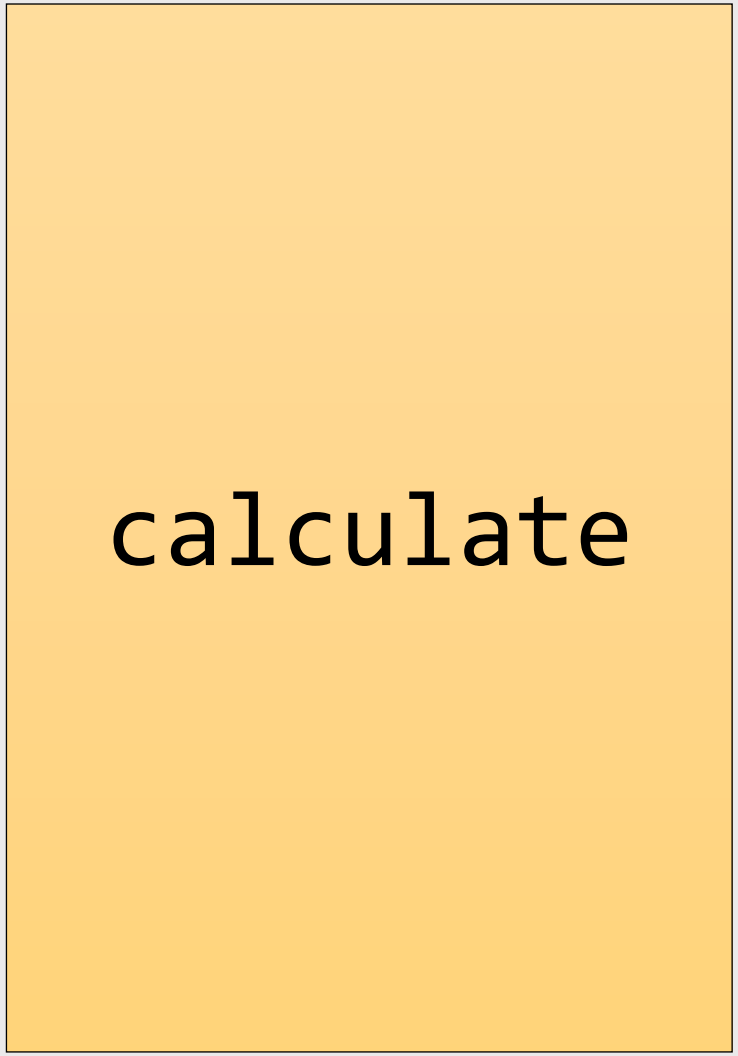
Call Stack

```
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}


int main()
{
    int result = calculate(10, 20);
    return 0;
}
```



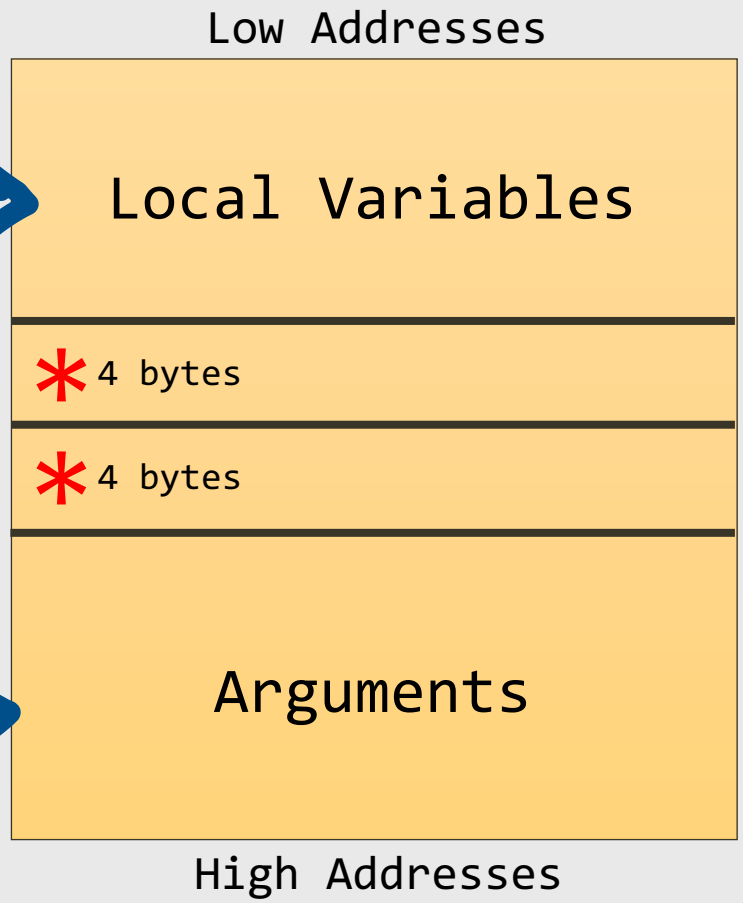
Stack Frame

```
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}
```



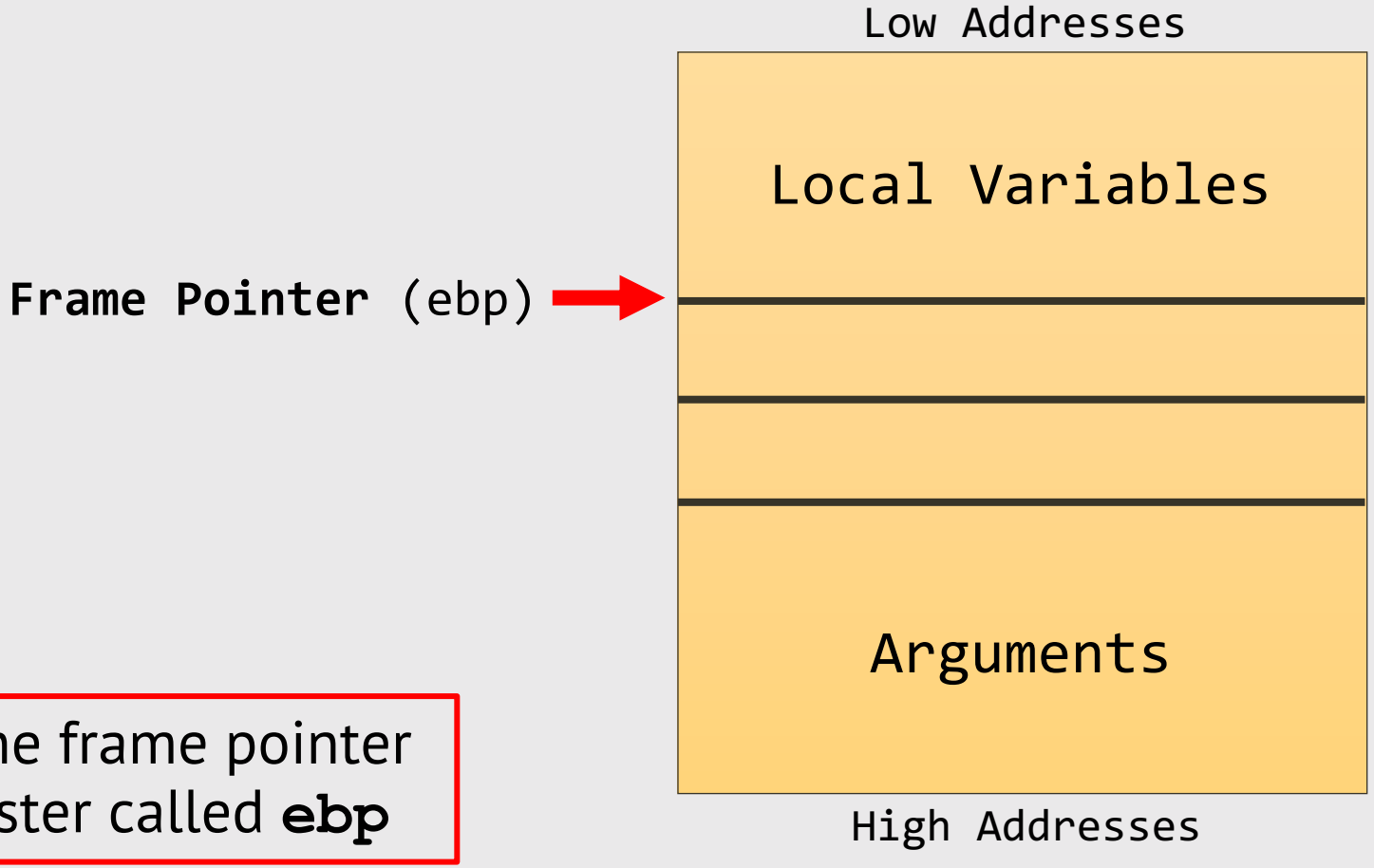
Stack Frame

```
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}
```



Stack Frame

```c
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}
```

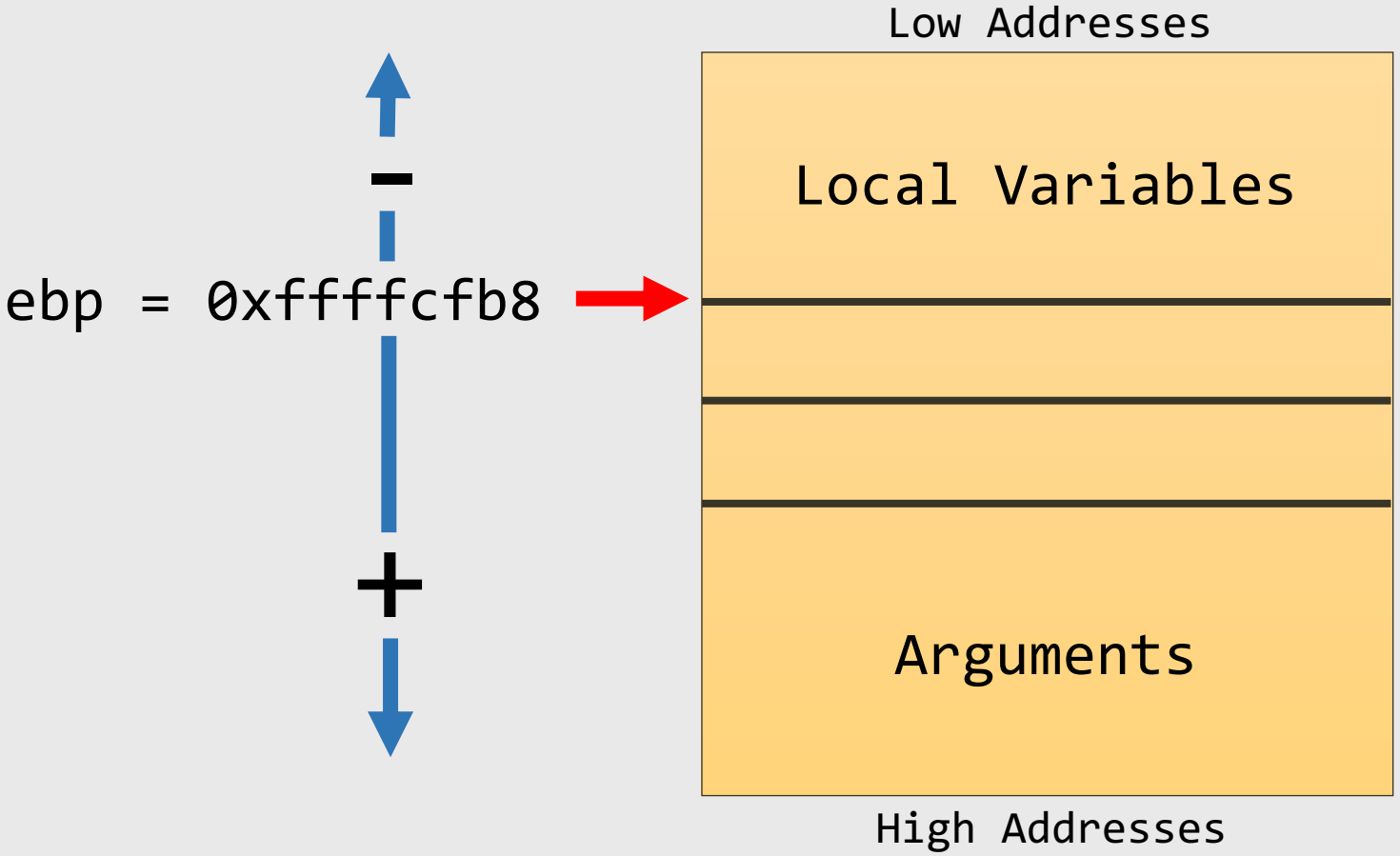


In the x86 architecture the frame pointer is stored in a special register called **ebp**

How do we know the addresses of the local variables and arguments?

Stack Frame

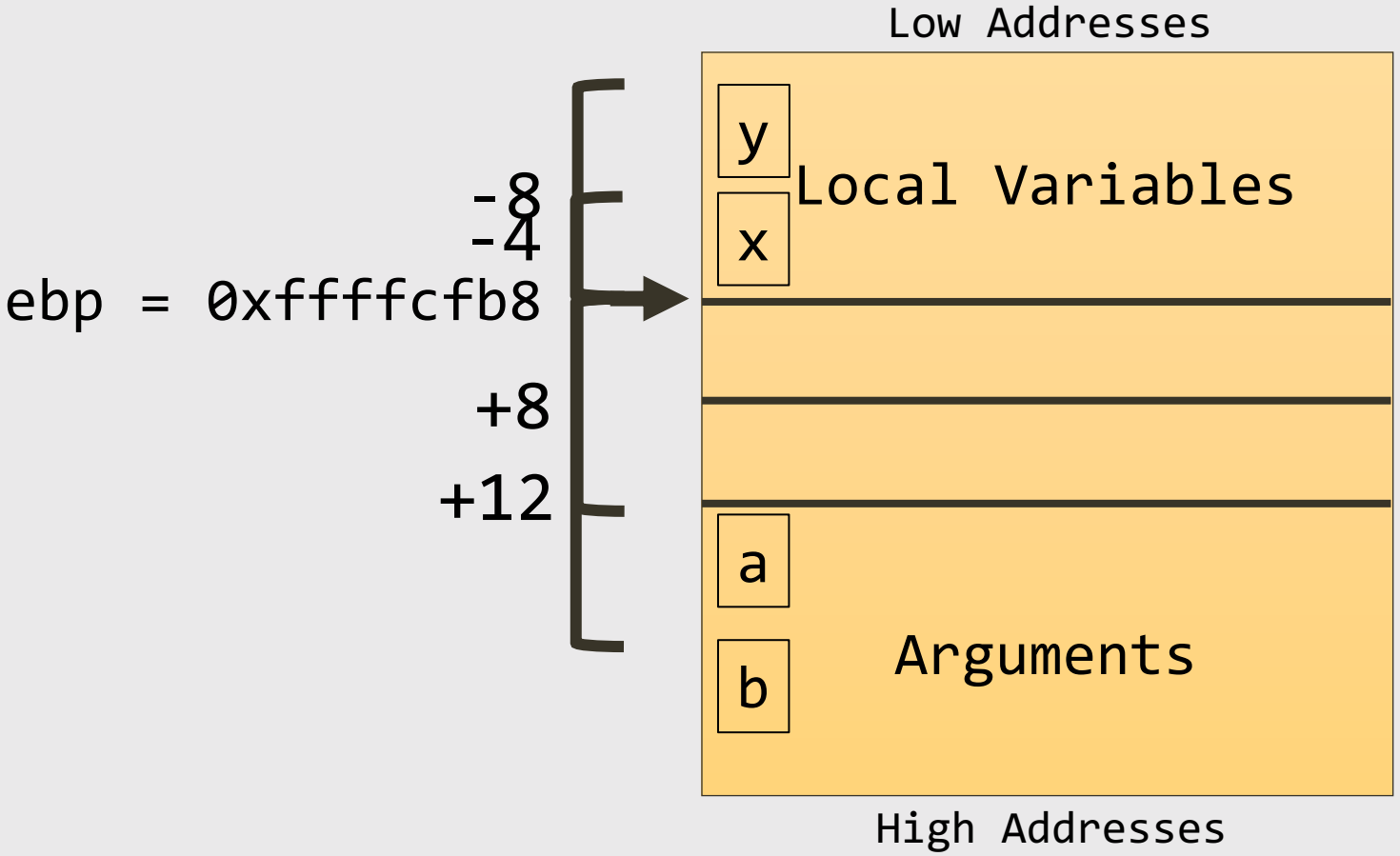
```
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}
```



We can add and subtract from the **frame pointer**

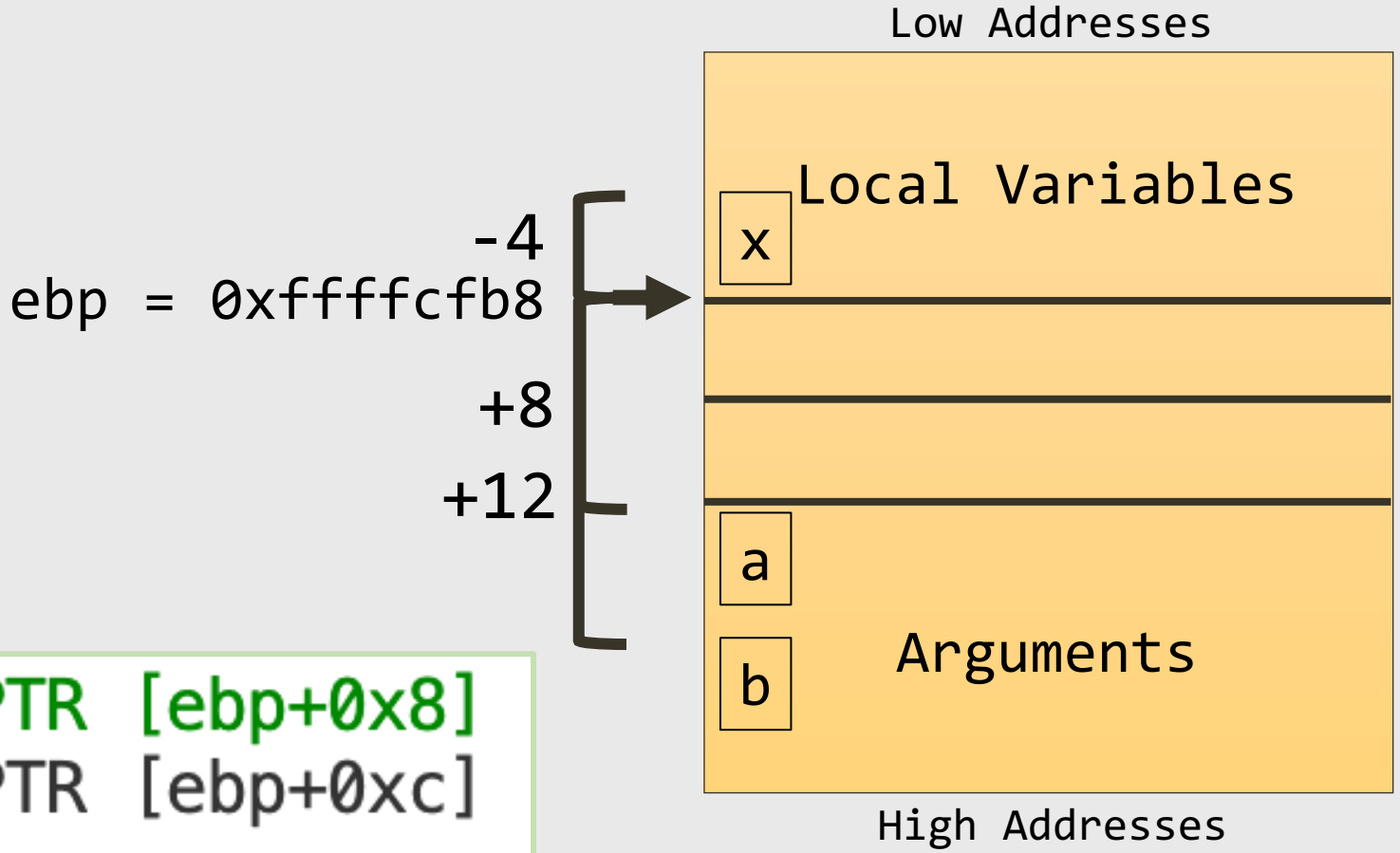
Stack Frame

```
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}
```



Stack Frame

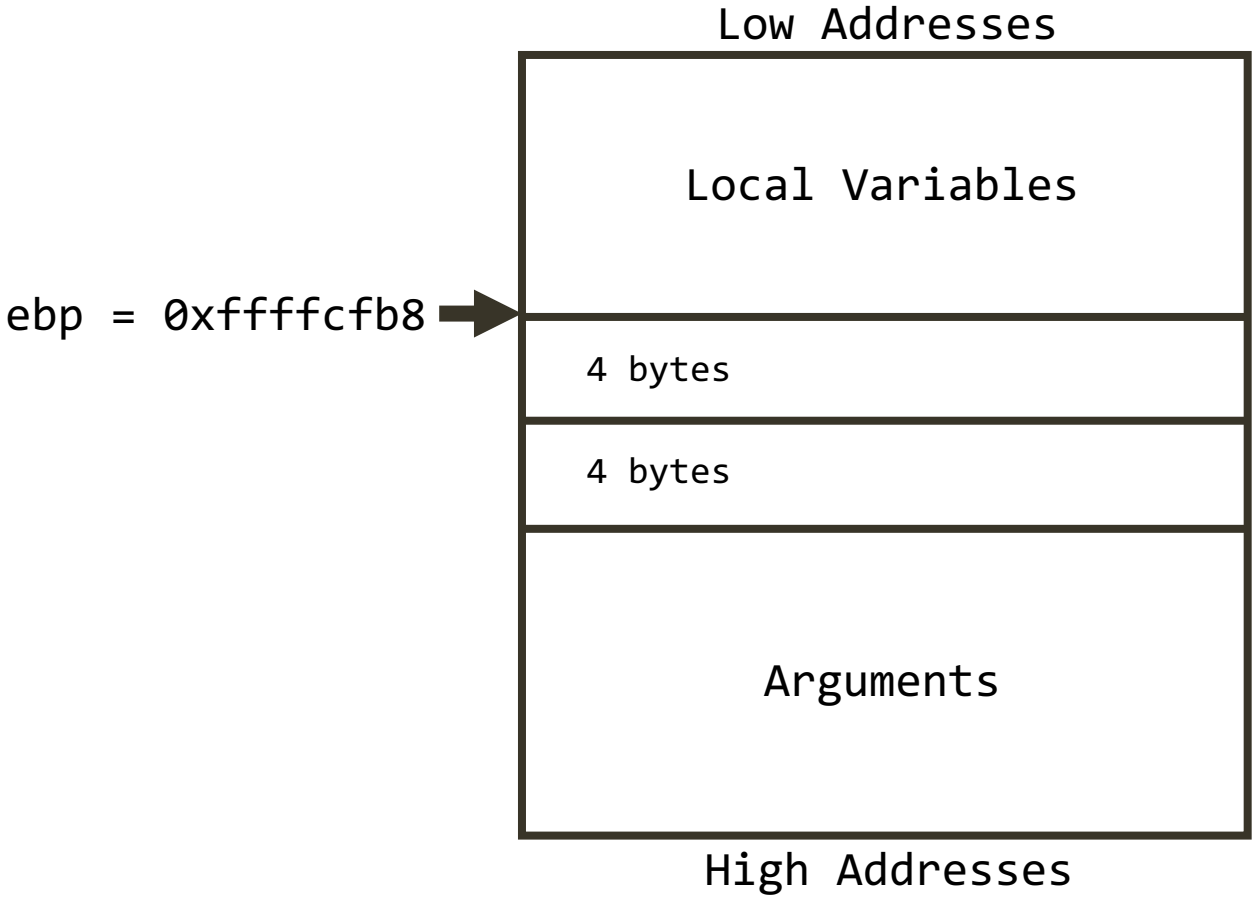
```c
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}
```




```asm
mov     edx, DWORD PTR [ebp+0x8]
mov     eax, DWORD PTR [ebp+0xc]
add     eax, edx
mov     DWORD PTR [ebp-0x4], eax
```

Write in the location of variables and arguments in the **stack frame**:

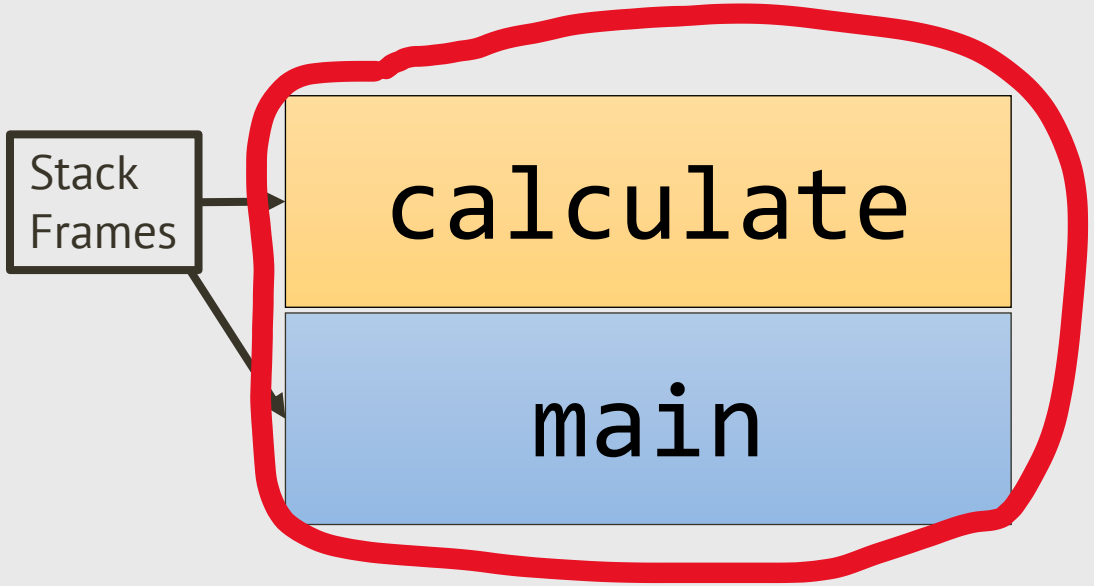
```c
int multiplier(int a, int b, int c)
{
    int x = 22;
    int y = 15;
    int z = 39;

    return (a * x) + (b * y) + (c * z);
}
```



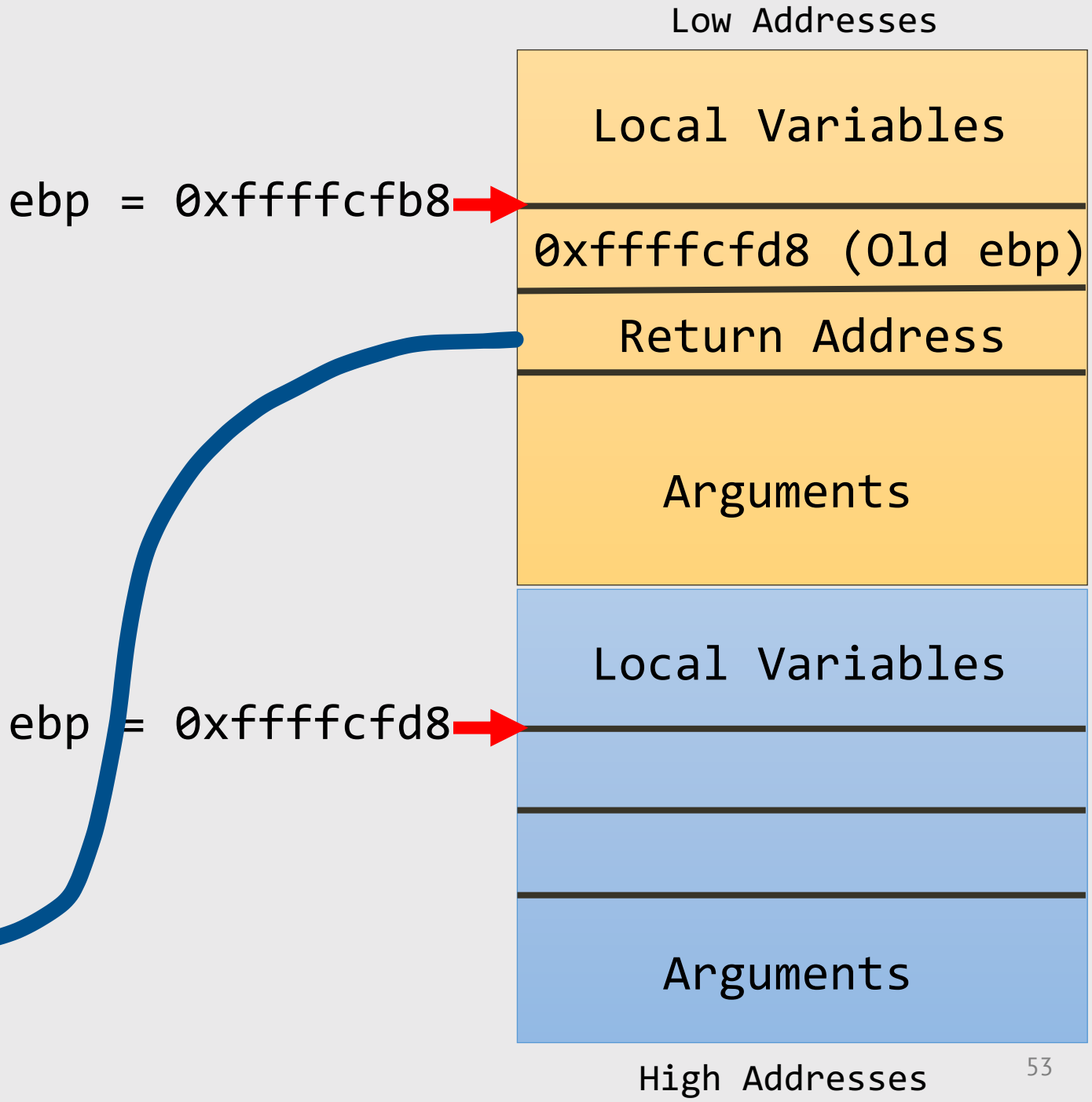
Write the distance in bytes from the **frame pointer** (ebp):

Call Stack

```
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}


int main()
{
    int result = calculate(10, 20);
    return 0;
}
```



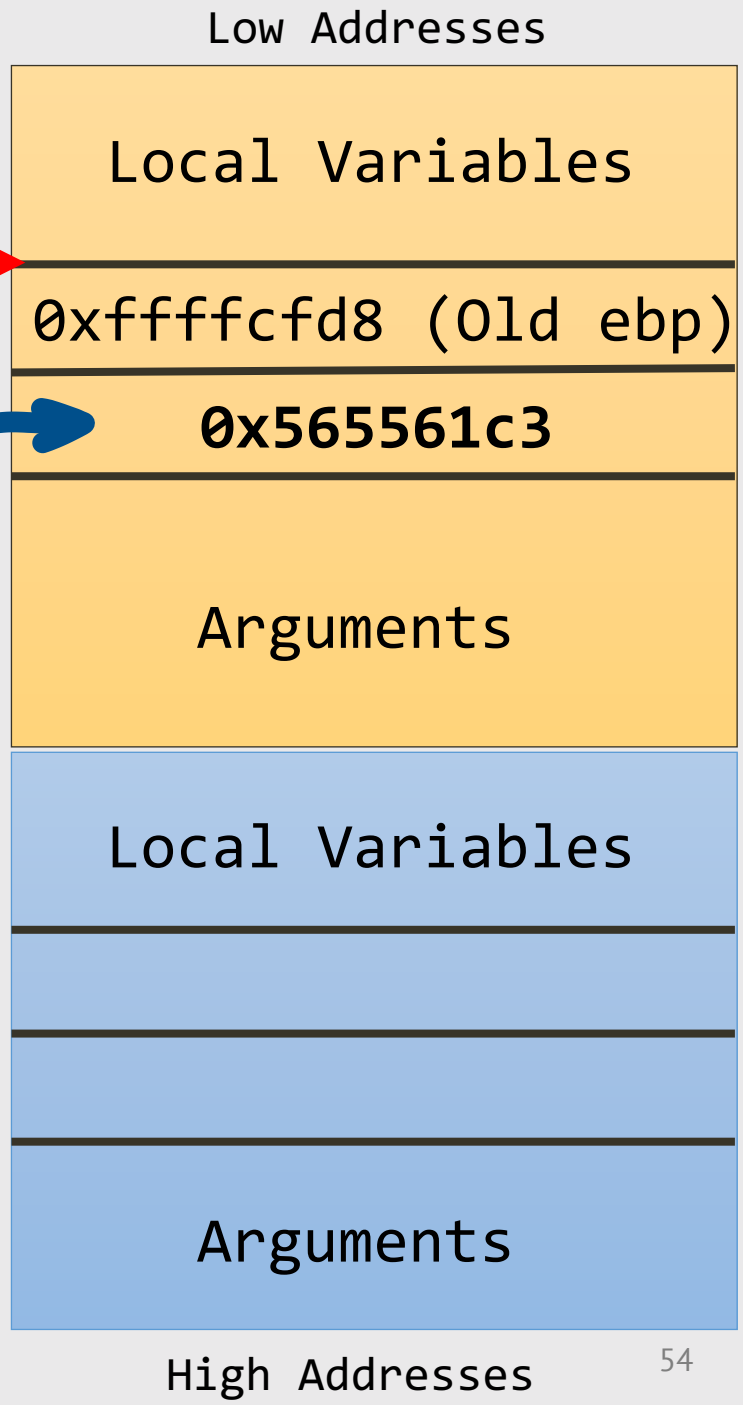
```c
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}


int main()
{
    calculate(10, 20);
    return 0;
}
```



```
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}


int main()
{
    calculate(10, 20);
    return 0;
}
```

ebp = 0xffffcfb8

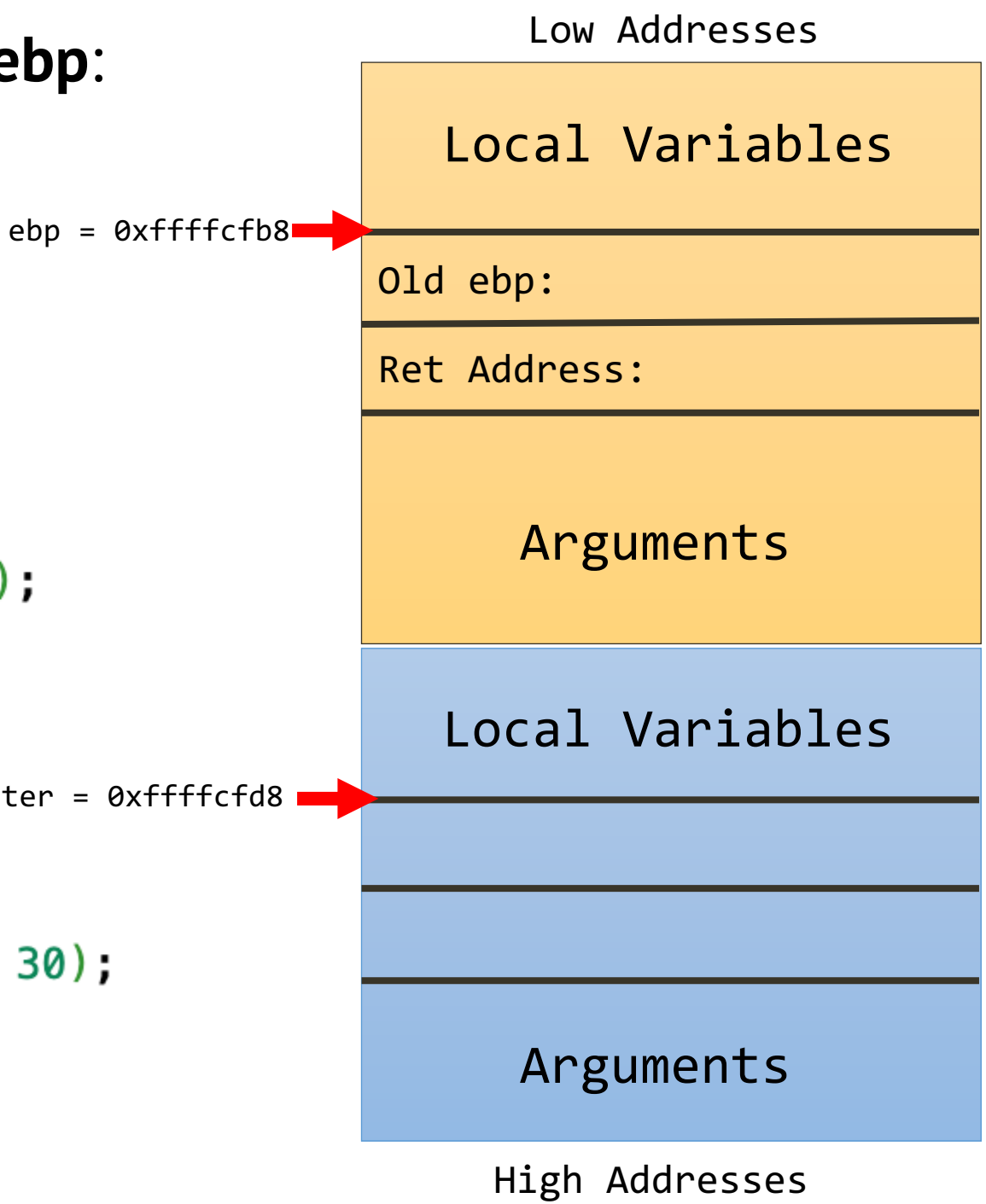
Return Address:
0x565561c3



Write in the **return address** and old **ebp**:

```
int multiplier(int a, int b, int c)
{
    int x = 22;
    int y = 15;
    int z = 39;

    return (a * x) + (b * y) + (c * z);
}


int main()
{
    int result = multiplier(10 , 20 , 30);

    return 0;
}
```

Return Address:
0x565561c3



Software Vulnerabilities

1. Introduction
2. Program memory layout
3. Stack layout
4. Buffer overflow vulnerability

```c
void copyInputToBuffer(char* input)
{
    char buffer[10];

    // Potential buffer overflow
    strcpy(buffer, input);
}

int main()
{
    printf("Enter a string: ");

    char input[256];
    fgets(input, sizeof(input), stdin);

    copyInputToBuffer(input);

    return 0;
}
```

```c
void copyInputToBuffer(char* input)
{
    char buffer[10];

    // Potential buffer overflow
    strcpy(buffer, input);
}

int main()
{
    printf("Enter a string: ");

    char input[256];
    fgets(input, sizeof(input), stdin);

    copyInputToBuffer(input);

    return 0;
}
```