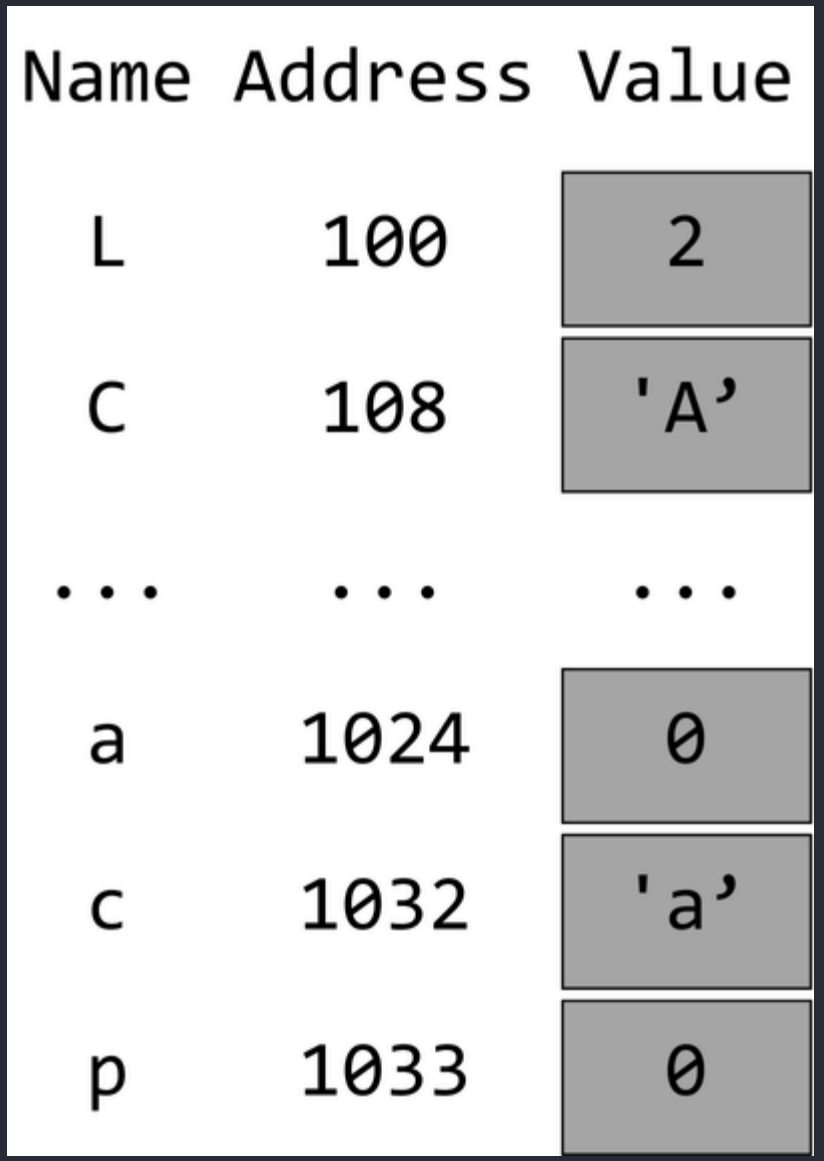
CS240 - Data Structures

Winter 2024

Types & Variables

- Visualize variables as blocks in memory
- Variable properties
  - Address, Size, Value
- Size
  - Associated with its type
  - Determines offset among addresses

```
1 long L = 2;
2 const char C = 'A';
3 int main() {
4     int a = 0;
5     const char c = 'a';
6     char* p = NULL;
7 }
```


Name Address Value
L 100 2
C 108 'A’
... ...
a 1024 0
c 1032 'a’
p 1033 0

`sizeof` operator

- Primitive types have their size defined at compile time
- `sizeof` operator returns size of type/variable in bytes

```
 1 int main() {
 2         char a = '0';
 3         printf("%d ", sizeof(a));
 4         printf("%d ", sizeof(char));
 5         printf("%d ", sizeof(int));
 6         printf("%d ", sizeof(bool));
 7         printf("%d ", sizeof(double));
 8         printf("%d ", sizeof(char*));
 9         printf("%d ", sizeof(long*));
10         printf("%d ", sizeof(void*));
11 }
```

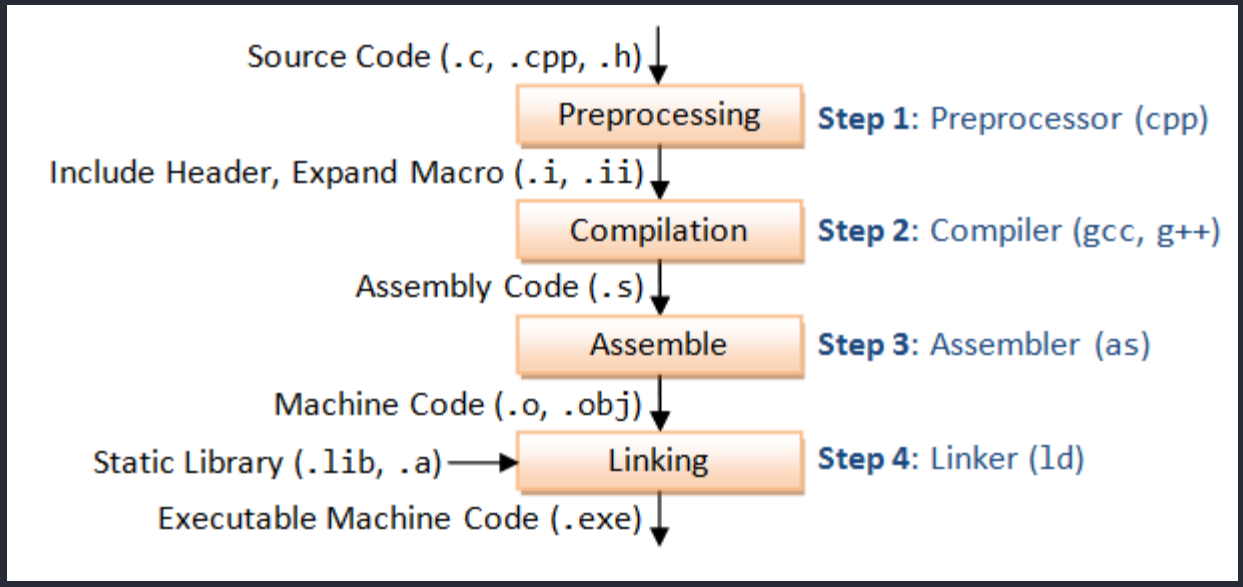
```
1 1 4 1 8 8 8 8
```

Compiling

```
gcc -g -O0 -std=c99 -Wall -Wextra -Wpedantic -o main main.c
```

- `-g` debug symbols
- `-O0` disable optimizations
- `-std=c99` pick language standard ISO C99
- `-Wall` warnings
- `-Wextra` more warnings
- `-Wpedantic` strict ISO C compliance warnings

Compile Chain



- **Preprocessor** parses `#include` and `#define` directives
- **Compiler** with assembler produce a machine instruction object file `.o` from a source code `.c` file
- **Linker** combines object files to a single executable `.out/.exe`

Debugging

Live Demo

Operators

```c
int main() {
    int a = 3 + 2;
    int b = 3 - 1;
    int c = a % b;
    printf("%d\n", c);
    int d = c++;
    printf("%d\n", d);
    d = --c;
    printf("%d\n", d);
    bool e = a < c;
    if (e) printf("less\n");
    else if (a == c) printf("equal\n");
    else printf("greater equal\n");
}
```

```
1
1
1
greater equal
```

- Visualize each operator as a function
  - Takes one/two arguments
  - Returns a value
  - May modify the state of its arguments

Operator	Type x	Type y	Type Return	Detail
x < y	numerical	numerical	boolean	x,y share Type
x == y	any	any	boolean	x,y share Type
++x	integer		integer	modifies x

Precedence & Associativity

- Precedence
  - Which is evaluated first
- Associativity
  - On same precedence, evaluate Left-to-Right or Right-to-Left

```
1 int main() {
2     int a = 3 * 2 + 1; // Precedence
3     int b = 6;
4     int c = 5;
5     a = b = c = 10; // Associativity Right-to-Left
6 }
```

Operator Precedence Table (Link)

Scope

- Brackets can cause scope change
- We can only access variables of active scopes
- Global scope is always active

```
 1 const bool c = false;
 2 int main() {
 3     int a = 1;
 4     {
 5         int b = 2;
 6         {
 7             int c = 3; // shadowing
 8             printf("%d %d %d\n", a, b, c);
 9         }
10     }
11     // a = b; //compiler error
12 }
```

```
1 2 3
```

enum-typedef

- An `enum` variable can hold one of predefined values
- Keyword `typedef` used to alias a type name

```c
enum Color {
    Red,
    Green,
    Blue,
};
typedef enum Color Color;
int main() {
    printf("%d\n", sizeof(Color));
    Color c = Red;
    // enum Color c = Red; //Syntax without typedef
    if (c == Red)           printf("Red\n");
    else if (c == Green)    printf("Green\n");
    else if (c == Blue)     printf("Blue\n");
    else                    printf("Error\n");
}
```

```
4
Red
```

Switch

`if/else` syntax suitable for `enums/ints`

```c
enum Color {
    Red,
    Green,
    Blue,
};
typedef enum Color Color;
int main() {
    Color c = Green;
    switch (c) {
        case Red:   printf("Red\n");    break;
        case Green: printf("Green\n");  break;
        case Blue:  printf("Blue\n");   break;
        default:    printf("Error\n");
    }
}
```

```
Green
```

Loops

```c
int main() {
    const int limit = 10;
    int i = 0;
    while (i < limit) {
        printf("%d ", i);
        i += 1;
    }
    printf("\n");
    for (int i = 0; i < limit; ++i) printf("%d ", i);
    printf("\n");
}
```

```
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
```

```c
int main() {
    int j = 3;
    bool isDone = false;
    while(!isDone) {
        printf("[%d %d] ", j, isDone);
        if (j > 0) { --j; continue; }
        isDone = true;
    }
    printf("\n");
    for(;;) {
        printf("before break\n");
        break;
    }
}
```

```
[3 0] [2 0] [1 0] [0 0]
before break
```

Functions

Reusable pieces of code

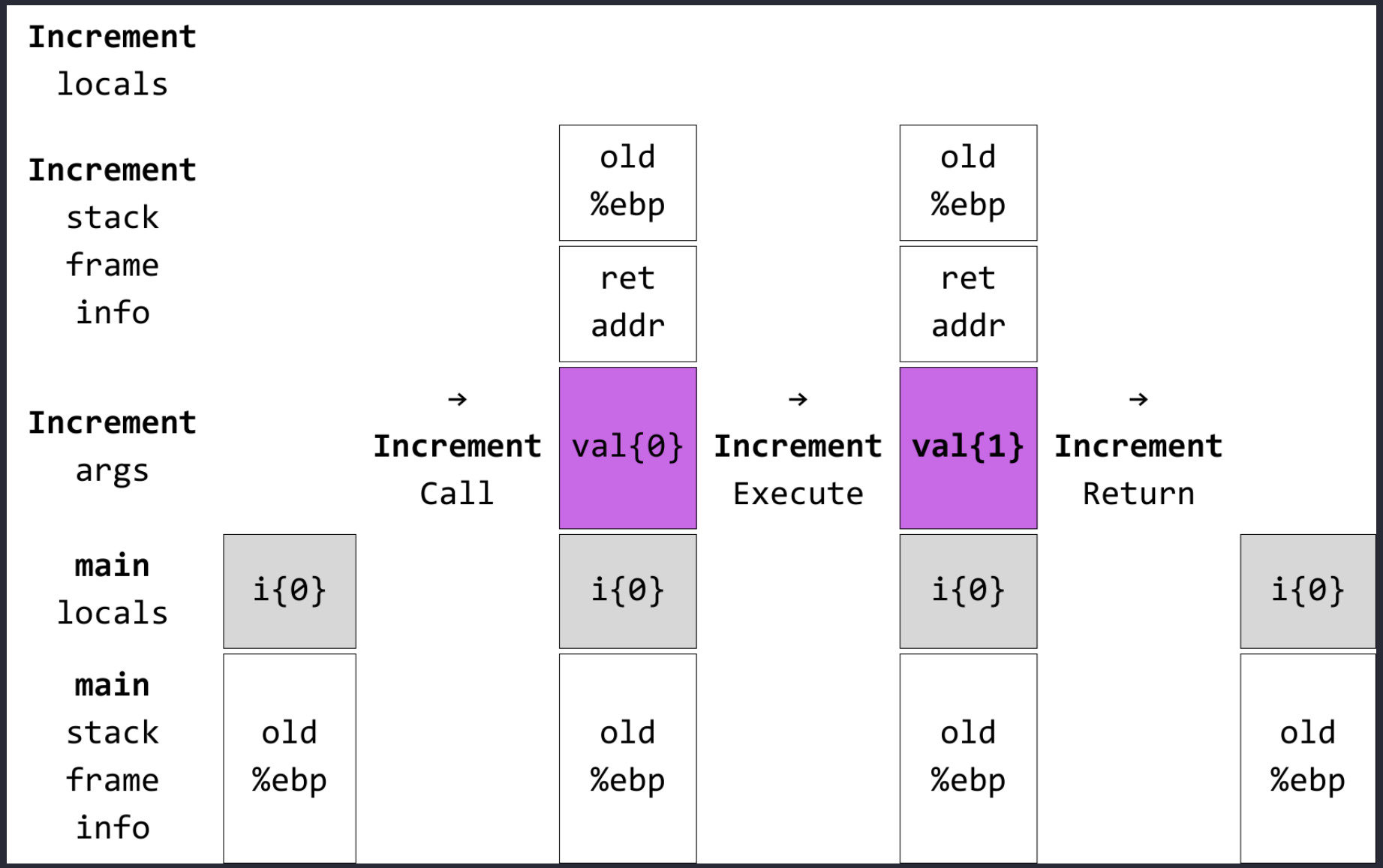
```
1 bool IsEven(int val) {return (val % 2) == 0;}
2 void PrintEven(int val, bool isEven) {
3     if(isEven) printf("%d is Even\n", val);
4 }
5 int main() {
6     for (int i = 0; i < 5; ++i)
7         PrintEven(i, IsEven(i));
8 }
```

```
0 is Even
2 is Even
4 is Even
```

Call-By-Value

```c
void Increment(int val) {
    val += 1;
    printf("%d\n", val);
}
int main() {
    int i = 0;
    printf("%d\n", i);
    Increment(i);
    printf("%d\n", i);
}
```

```
0
1
0
```


Increment
locals
Increment
stack
frame
info
Increment
args
main
locals
main
stack
frame
info
i{0}
old
%ebp
→
Increment
Call
old
%ebp
ret
addr
val{0}
i{0}
old
%ebp
→
Increment
Execute
old
%ebp
ret
addr
val{1}
i{0}
old
%ebp
→
Increment
Return
i{0}
old
%ebp

Recursion

```c
int Pow(int base, int exp) {
    if (exp == 0) return 1;
    return base * Pow(base, exp - 1);
}
int main() {
    printf("%d\n", Pow(2, 10));
}
```

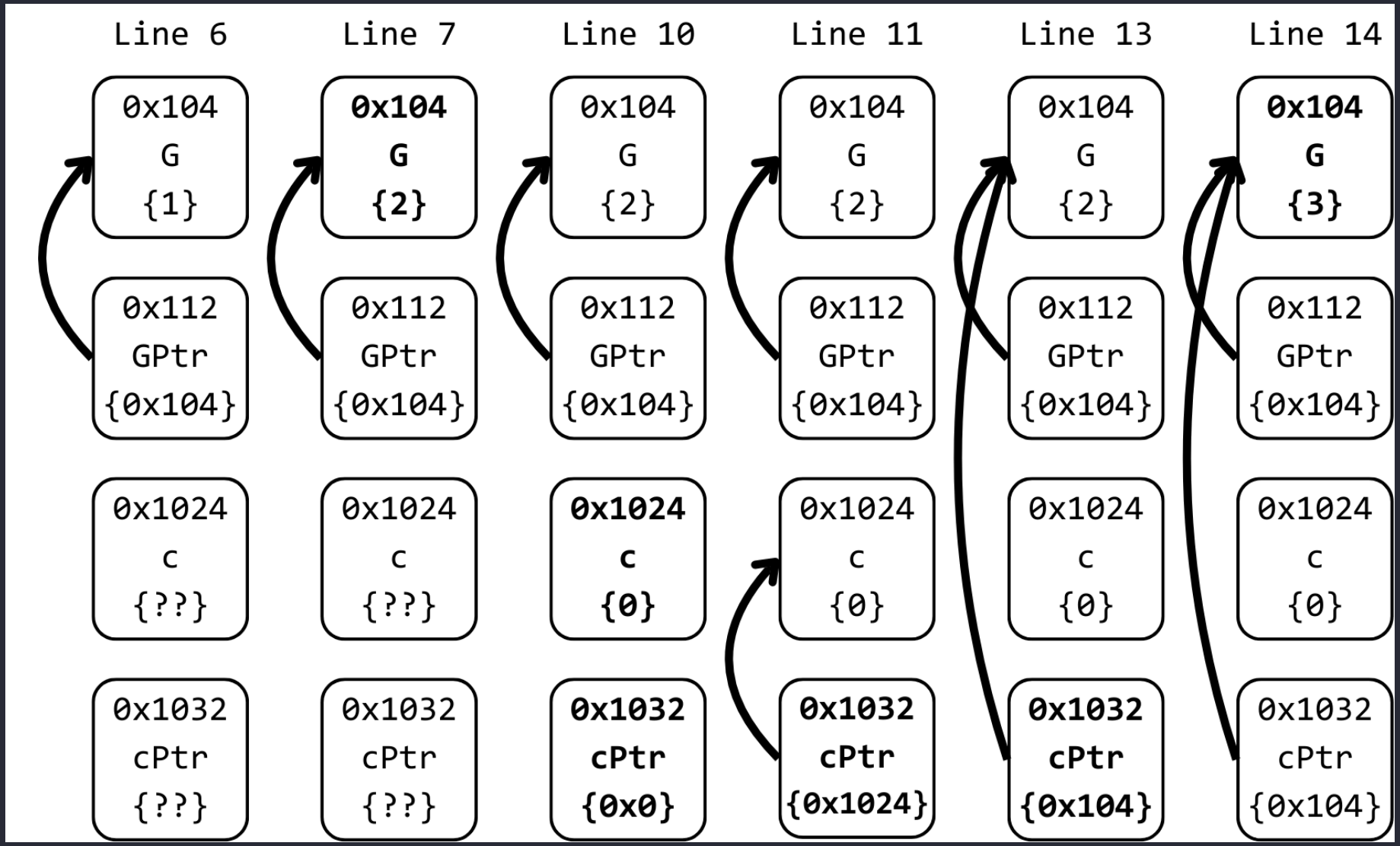
```
1024
```

Pointers

The value of a pointer variable is a memory address

```
 1 int G = 1;
 2 int* const GPtr = &G;
 3 int main() {
 4     printf("%d ", sizeof(int*));
 5     printf("%d\n", sizeof(char*));
 6     *GPtr = 2;
 7     printf("%p %d %p %d\n", &G, G, GPtr, *GPtr);
 8     int c = 0;
 9     int* cPtr = NULL;
10     cPtr = &c;
11     printf("%p %d %p %d\n", &c, c, cPtr, *cPtr);
12     cPtr = GPtr;
13     *cPtr += 1;
14     printf("%p %d %p %d\n", &G, G, GPtr, *GPtr);
15 }
```

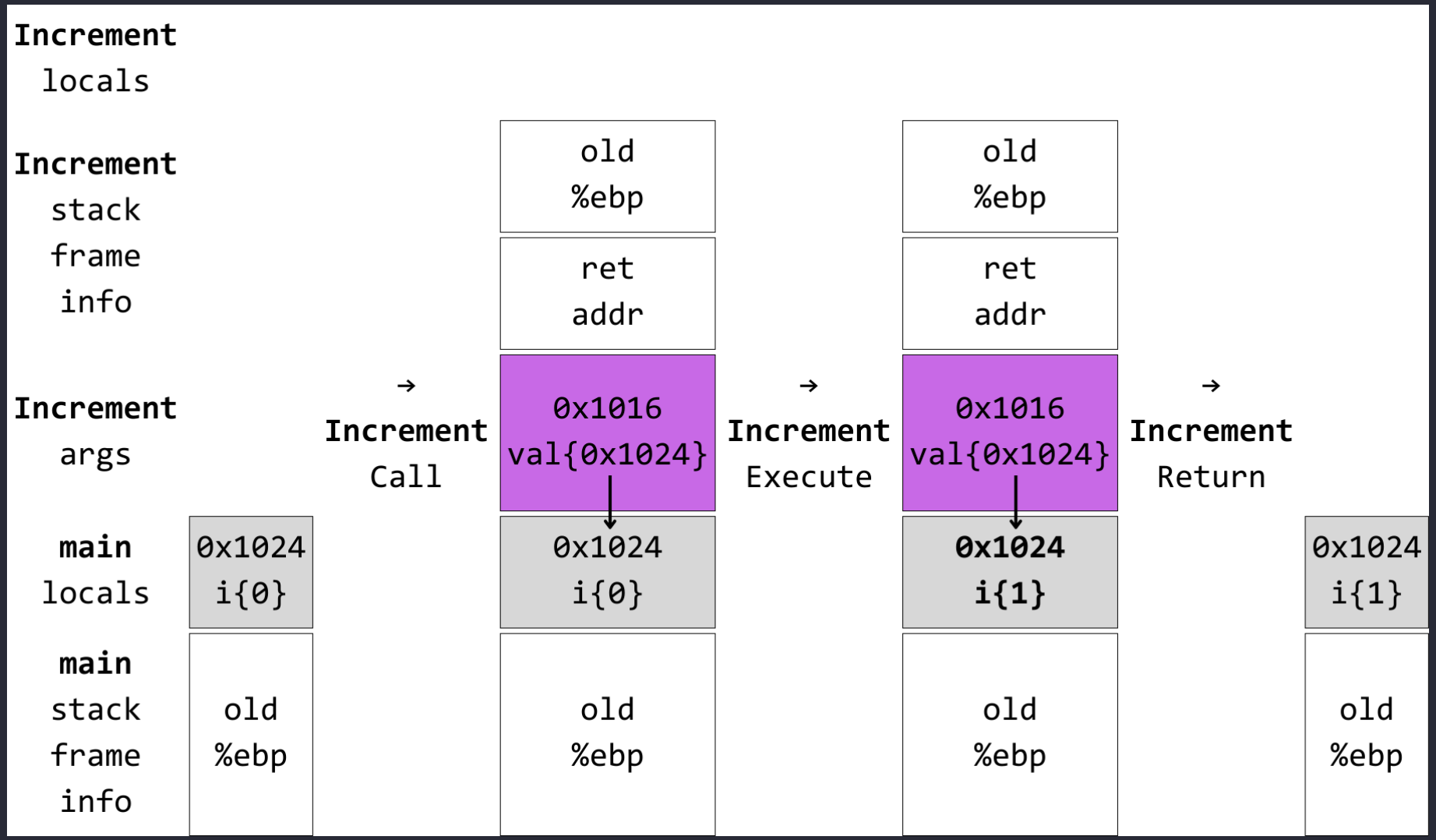
```
8 8
0×404018 2 0×404018 2
0×7ffdf15d3904 0 0×7ffdf15d3904 0
0×404018 3 0×404018 3
```


Line 6
Line 7
Line 10
Line 11
Line 13
Line 14
0x104 G {1}
0x112 GPtr {0x104}
0x1024 c {??}
0x1032 cPtr {??}
0x104 G {2}
0x112 GPtr {0x104}
0x1024 c {??}
0x1032 cPtr {??}
0x104 G {2}
0x112 GPtr {0x104}
0x1024 c {0}
0x1032 cPtr {0x0}
0x104 G {2}
0x112 GPtr {0x104}
0x1024 c {0}
0x1032 cPtr {0x1024}
0x104 G {2}
0x112 GPtr {0x104}
0x1024 c {0}
0x1032 cPtr {0x104}
0x104 G {3}
0x112 GPtr {0x104}
0x1024 c {0}
0x1032 cPtr {0x104}

Call-By-Reference Trick

```
 1 void Increment(int* val) {
 2     *val += 1;
 3     printf("%d\n", *val);
 4 }
 5 int main() {
 6     int i = 0;
 7     printf("%d\n", i);
 8     Increment(&i);
 9     printf("%d\n", i);
10 }
```

```
0
1
1
```


Increment
locals
Increment
stack
frame
info
Increment
args
main
locals
main
stack
frame
info
0x1024
i{0}
old
%ebp
Increment
Call
old
%ebp
ret
addr
0x1016
val{0x1024}
0x1024
i{0}
old
%ebp
Increment
Execute
old
%ebp
ret
addr
0x1016
val{0x1024}
0x1024
i{1}
old
%ebp
Increment
Return
0x1024
i{1}
old
%ebp

Structs

Definition of new type composed by simpler types

```c
struct Pair {
    int x;
    int y;
};
typedef struct Pair Pair;
int main() {
    printf("%d\n", sizeof(Pair));
    Pair pair1; //members uninitialized
    pair1.x = 1; pair1.y = 2;
    printf("%d %d\n", pair1.x, pair1.y);
    Pair pair2 = {.x=3, .y=4};
    printf("%d %d\n", pair2.x, pair2.y);
    Pair pair3 = pair2;
    printf("%d %d\n", pair3.x, pair3.y);
}
```

```
8
1 2
3 4
3 4
```

C-Arrays

Size must be known at compile time

```c
struct Pair {
    int x;
    int y;
};
typedef struct Pair Pair;
#define PAIRS_SIZE 3
int main() {
    Pair pairs1[PAIRS_SIZE]; //elements uninitialized
    printf("%d\n", sizeof(pairs1));
    for (int i = 0; i < PAIRS_SIZE; ++i)
        { pairs1[i].x=0; pairs1[i].y=0; }
    Pair pairs2[PAIRS_SIZE] = { {1, 1}, {1, 1}, {1, 1} };
    Pair pairs3[PAIRS_SIZE] = {}; //elements zero'ed
}
```

```
24
```

C-Arrays As Pointers

```c
#define N_SIZE 10
void PrintArray(int* arr) {
    for (int i = 0; i < N_SIZE; ++i) printf("%d ", arr[i]);
}
int main() {
    int n[N_SIZE] = {1, 2}; //rest zero'ed
    PrintArray(n); printf("\n");
    int* const nAlias = n;
    nAlias[0] = 3; n[1] = 4;
    *(nAlias + 2) = 5; *(n + 3) = 6;
    int* const nIdxFour = n + 4; *nIdxFour = 7;
    PrintArray(n); printf("\n");
    printf("%p %p %d\n",&(n[5]),(n+5),n[5]);
    printf("%p %p %d\n",&(nAlias[5]),(nAlias+5),nAlias[5]);
}
```

```
1 2 0 0 0 0 0 0 0 0
3 4 5 6 7 0 0 0 0 0
0×7ffdb2f8cd74 0×7ffdb2f8cd74 0
0×7ffdb2f8cd74 0×7ffdb2f8cd74 0
```

Dynamic Allocation

- `malloc` returns pointer to newly allocated, uninitialized memory
- `free` deallocates memory

```c
int main() {
    int* numPtr = NULL;
    numPtr = malloc(1 * sizeof(int));
    *numPtr = 1;
    *(numPtr + 0) = 2;
    int num = *numPtr;
    printf("%p %d %p %d\n", numPtr, *numPtr, &num, num);
    free(numPtr);
}
```

```
0×6df2a0 2 0×7ffed055c8a4 2
```

Dynamic Allocation Of Struct

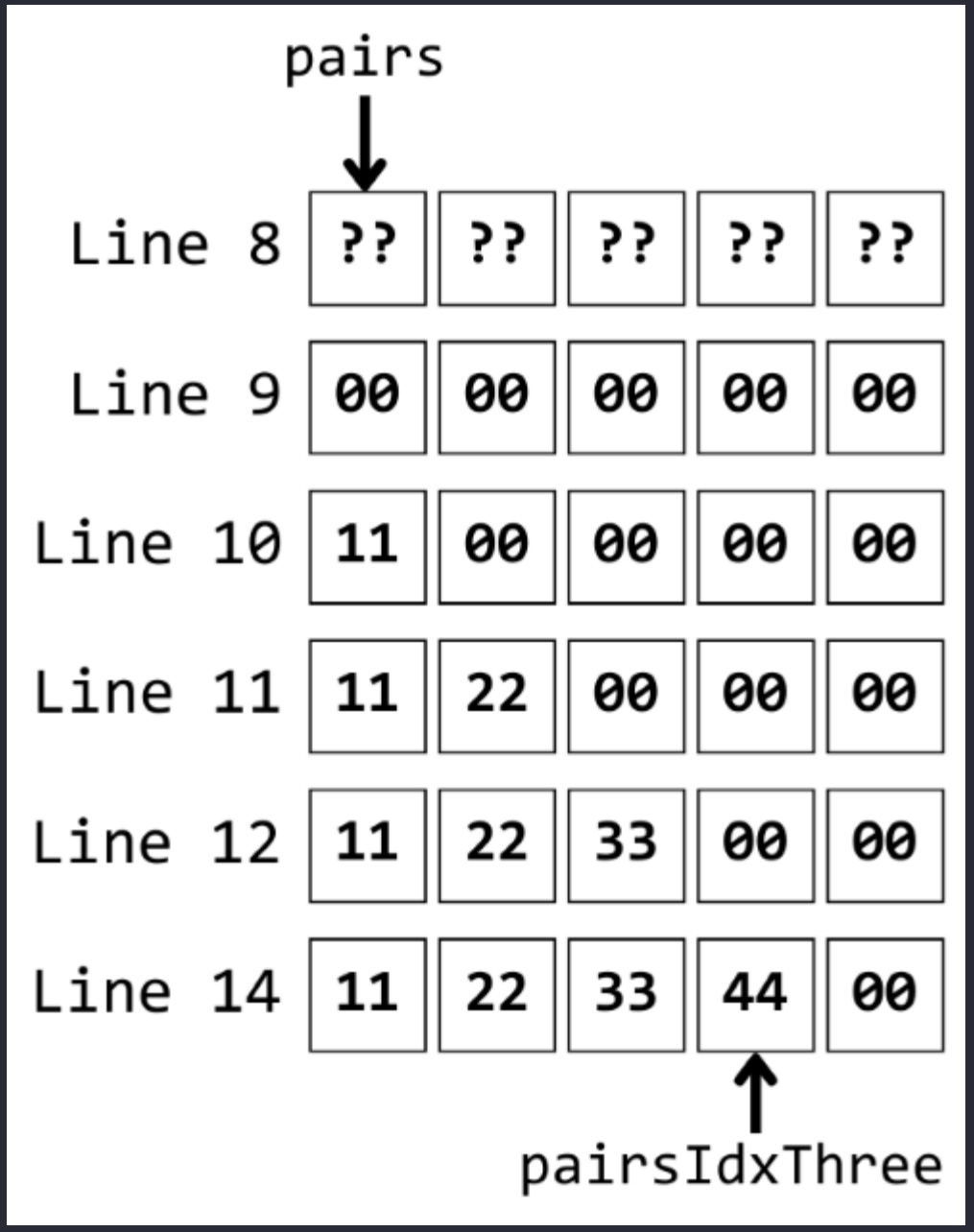
```c
struct Pair { int x; int y; };
typedef struct Pair Pair;
int main() {
    Pair* pair = malloc(sizeof(Pair));
    (*pair).x = (*pair).y = 1;
    pair->x = pair->y = 2;
    Pair temp = {.x=3, .y=3}; *pair = temp;
    pair->x = pair->y = 4;
    Pair result = *pair;
    printf("%p %d %d\n", pair, pair->x, pair->y);
    printf("%p %d %d\n", result, result.x, result.y);
    printf("%p %d %d\n", temp, temp.x, temp.y);
    free(pair);
}
```

```
0×f8c2a0 4 4
0×400000004 4 4
0×300000003 3 3
```

Dynamic Allocation Of Struct-Array

```
 1 typedef struct Pair { int x; int y; } Pair;
 2 const int PairsSize =  5;
 3 void PrintPairs(Pair* arr) {
 4     for (int i = 0; i < PairsSize; ++i)
 5         printf("%d%d ", arr[i].x, arr[i].y); }
 6 int main() {
 7     Pair* pairs = malloc(PairsSize * sizeof(Pair));
 8     memset(pairs, 0, PairsSize * sizeof(Pair));
 9     pairs[0].x = pairs[0].y = 1;
10     (*(pairs + 1)).x = (*(pairs + 1)).y = 2;
11     (pairs + 2)->x = (pairs + 2)->y = 3;
12     Pair* pairsIdxThree = pairs + 3;
13     pairsIdxThree->x = pairsIdxThree->y = 4;
14     PrintPairs(pairs); printf("\n");
15     free(pairs); }
```

```
11 22 33 44 00
```


pairs
Line 8
?? ?? ?? ?? ??
Line 9
00 00 00 00 00
Line 10
11 00 00 00 00
Line 11
11 22 00 00 00
Line 12
11 22 33 00 00
Line 14
11 22 33 44 00
pairsIdxThree

Defining a Singly-Linked List

```
 1 struct PlayerNode {
 2     int id;
 3     int score;
 4     struct PlayerNode* next;
 5 };
 6 typedef struct PlayerNode PlayerNode;
 7 struct PlayerList {
 8     PlayerNode* head;
 9 };
10 typedef struct PlayerList PlayerList;
```

```
PlayerList* PlayerList_Create() {
    PlayerList* list = malloc(sizeof(PlayerList));
    list→head = NULL;
    return list;
}
void PlayerList_Destroy(PlayerList* list) {
    PlayerNode *prev = NULL;
    for (PlayerNode* p=list→head; p≠NULL; p=p→next) {
        free(prev);
        prev = p;
    }
    free(prev);
}
```

```c
void PlayerList_Insert(PlayerList* list, int id) {
    PlayerNode* node = malloc(sizeof(PlayerNode));
    node->id = id;
    node->score = 0;
    node->next = NULL;

    node->next = list->head;
    list->head = node;
}
```

```c
void PlayerList_Print(PlayerList* list) {
    for (PlayerNode* p = list->head; p!=NULL; p=p->next)
        printf("[%p %d %d %p]\n",p,p->id,p->score,p->next);
}
int main() {
    PlayerList* list = PlayerList_Create();
    PlayerList_Insert(list, 10);
    PlayerList_Insert(list, 20);
    PlayerList_Print(list);
    PlayerList_Destroy(list);
}
```

```
[0x22662e0 20 0 0x22662c0]
[0x22662c0 10 0 (nil)]
```

- 🧐 Provide feedback and report slide errors 🧐
- 🤖 A.I. generated code is strictly prohibited 🤖

Thank you