Coding guidelines, advanced C and debugging

Ignas Brazauskas

https://github.com/Ignas207/robotics-materials

Notes from last lecture

- Strings are surounded by `"`
  - e.g. `char testMsg[] = "Hello!\n"`
- `'` denotes a ***single* character!**
  - e.g. `char test1 = 'A'`
- This is will **cause an error**:
  - `char thisIsProblematic[] = 'Hello!\n'`
- `\n` is a **newline**
  - This means, text after it will be printed in a **new line**!

Notes from last lecture

- For `#define`, we do it **at the top of our `.c` file!**

```c
#include <stdio.h>

#define ARRAY_SIZE 8 // The define is at the TOP!

int main(void)
{
    int numbers[] = {1, 2, 4, 8, 16, 32, 64, 128};
    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        printf("%d, ", numbers[i]);
    }
    printf("\n");
    return 0;
}
```

The best part about the C language

- Simple, straight forward
- Only a few rules to follow
- **You can do anything!**

Worst part about the C language

- You can do anything!?

Biggest problem in software development?

- You have to read **your code** a week from now!

An even *bigger problem*!?

- You have to read **other people's code**
- *And you have to work with them...*

Solution?

- We have to organize our code!

We need to comunicate our intent to the compiler

That's a programming language!

- Communicate **our intent** to *others*!

What do we mean by intent

- Make the code more **verbose**!
- *Why does this code exists?*

Calculator example: two versions

```c
#include <stdio.h>

int Multiply(int multiplicant, int multiplier);

int main(void)
{
    int firstMember, secondMember;
    printf("Enter multiplicant: ");
    scanf("%d", &firstMember);

    printf("Enter multiplier: ");
    scanf("%d", &secondMember);

    int result = Multiply(firstMember, secondMember);
    printf("Result is %d!\n", result);
    return 0;
}

int Multiply(int multiplicant, int multiplier)
{
    int result = multiplicant * multiplier;
    return result;
}
```

```c
#include <stdio.h>

int Multiply(int a, int b);

int main(void)
{
    int n, m;
    printf("Enter multiplicant: ");
    scanf("%d", &n);

    printf("Enter multiplier: ");
    scanf("%d", &m);

    int r1 = Multiply(n, m);
    printf("Result is %d!\n", r1);
    return 0;
}

int Multiply(int a, int b)
{
    int c = a * b;
    return c;
}
```

Name variables in a way that helps describe *their functionality, existance*!

- The code becomes **self documenting!**
  - *We can add comments with* `//` *or* `/* */`

#define directive

- Make the code more readable by eliminating **magic numbers!**
- `#define` just **replaces our text** in place!

```
for (int i = 0; i < 10; i++)
{
    // do something ...
}
```

```
#define MAX_COUNTER 10

for (int i = 0; i < MAX_COUNTER; i++)
{
    // do something ...
}
```

- These will compile the same result!
- *Define also documents the code!*

General coding guidelines

- *These are very general*
- **Variable** names are **lowerCase**
  - e.g. `int fistMember`, `float distance`
- **Function** names are **UpperCase**
  - e.g. `Calculate()`, `FindDistance()`
- **Contants** and `#define`'s are **IN_CAPS**
  - e.g. `MAX_MEMBERS`, `DELAY_AMOUNT_MS`

Other coding guidelines

- They are other code formating styles
  - Other projects, companies, etc... **can have their specifications**!
  - *You will have to follow them!*
- The most important part: **be consistent!**
- Find a **style witch suites you!**

Structures

Its an object!

Like a box?

Grouping datatypes with struct

- Allows us to create "objects" *by putting relevant members in the same box!*
- Usualy are *typedef'd*
  - *i.e. it's treated like a type*

```
typedef struct
{
    [type] [members name];
    [type] [members name];
    // etc...
} [StructName];
```

```
typedef struct
{
    char name[50];
    unsigned int age;
    float weight;
} Person;
```

- Easier to pass **multiple variables** into functions!
- The same Structs mambers names **can't overlap!**

Structures: examples

Structures: examples

```c
// Find this file in `examples/date-structures/printing-date-struct.c`
#include <stdio.h>

// Struct definition
typedef struct
{
    int year;
    int month;
    int day;
} Date;

void PrintDate(Date date);

int main(void)
{
    // Struct creation
    Date usersDate;
    // Accesing members
    usersDate.year = 2023;
    usersDate.month = 9;
    usersDate.day = 20;

    PrintDate(usersDate);
    return 0;
}

void PrintDate(Date date)
{
    printf("The year is %d, month is %d, today is %d.\n",
            date.year,
            date.month,
            date.day);
}

// Output: The year is 2023, month is 9, today is 20.
```

```c
// Find this file in `examples/date-structures/
//                    printing-date-struct-two-members.c`
int main(void)
{
    // Struct creation
    Date usersDate;
    Date davesDate;
    // Accesing members
    usersDate.year = 2023;
    usersDate.month = 9;
    usersDate.day = 20;

    davesDate.year = 1990;
    davesDate.month = 11;
    davesDate.day = 14;

    PrintDate(usersDate);
    PrintDate(davesDate);
    return 0;
}
// Output: The year is 2023, month is 9, today is 20.
//         The year is 1990, month is 11, today is 14.
```

Variables with an exact size

- Sometimes, it is beneficial to have variables that are an **exact guaranteed size**
- We can `#include <stdint.h>`
- `uint8_t` - unsigned integer with a size of **8 bits**
- `int8_t` - signed integer with a size of **8 bits**
- `uint16_t` - unsigned integer with a size of **16 bits**
- `uint32_t` - unsigned integer with a size of **32 bits**

Device handles

Using devices with less confusion!

- Contain all the information in an easy to use form!

Structures as device handles

```
/** @defgroup I2C_handle_Structure_definition I2C handle Structure definition
  * @brief  I2C handle Structure definition
  * @{
  */
typedef struct __I2C_HandleTypeDef
{
  I2C_TypeDef                *Instance;      /*!< I2C registers base address                */
  I2C_InitTypeDef            Init;          /*!< I2C communication parameters              */
  uint8_t                    *pBuffPtr;     /*!< Pointer to I2C transfer buffer            */
  uint16_t                   XferSize;      /*!< I2C transfer size                         */
  __IO uint16_t              XferCount;     /*!< I2C transfer counter                      */
  __IO uint32_t              XferOptions;   /*!< I2C sequantial transfer options, this parameter can
  be a value of @ref I2C_XFEROPTIONS */
  __IO uint32_t              PreviousState; /*!< I2C communication Previous state          */
  HAL_StatusTypeDef(*XferISR)(struct __I2C_HandleTypeDef *hi2c, uint32_t ITFlags, uint32_t ITSources);
  /*!< I2C transfer IRQ handler function pointer */
  DMA_HandleTypeDef          *hdmatx;       /*!< I2C Tx DMA handle parameters              */
  DMA_HandleTypeDef          *hdmarx;       /*!< I2C Rx DMA handle parameters              */
  HAL_LockTypeDef            Lock;          /*!< I2C locking object                        */
  __IO HAL_I2C_StateTypeDef  State;         /*!< I2C communication state                   */
  __IO HAL_I2C_ModeTypeDef   Mode;          /*!< I2C communication mode                    */
  __IO uint32_t              ErrorCode;     /*!< I2C Error code                            */
  __IO uint32_t              AddrEventCount; /*!< I2C Address Event counter                */
  __IO uint32_t              Devaddress;    /*!< I2C Target device address                 */
  __IO uint32_t              Memaddress;    /*!< I2C Target memory address                 */
} I2C_HandleTypeDef;
```

Now we get to the complex part

- *Pointers and addresess!*

What is a pointer?

- A pointer is a **variable** witch stores **an address** *or refference* to a variable.
- Denoted with a `*` symbol

```
int *number;
float *pi;
unsigned char *buff;
// etc ...
```

All of the variable naming rules apply here!

What is a pointer?

Analogy time!

- I mentioned an *address*?
- If you order something online and want that item to be **shipped to your house**, you put **your home address**.
- Your **home address** describes the ***physical location*** of your house.

Okay, how does this relate to C?

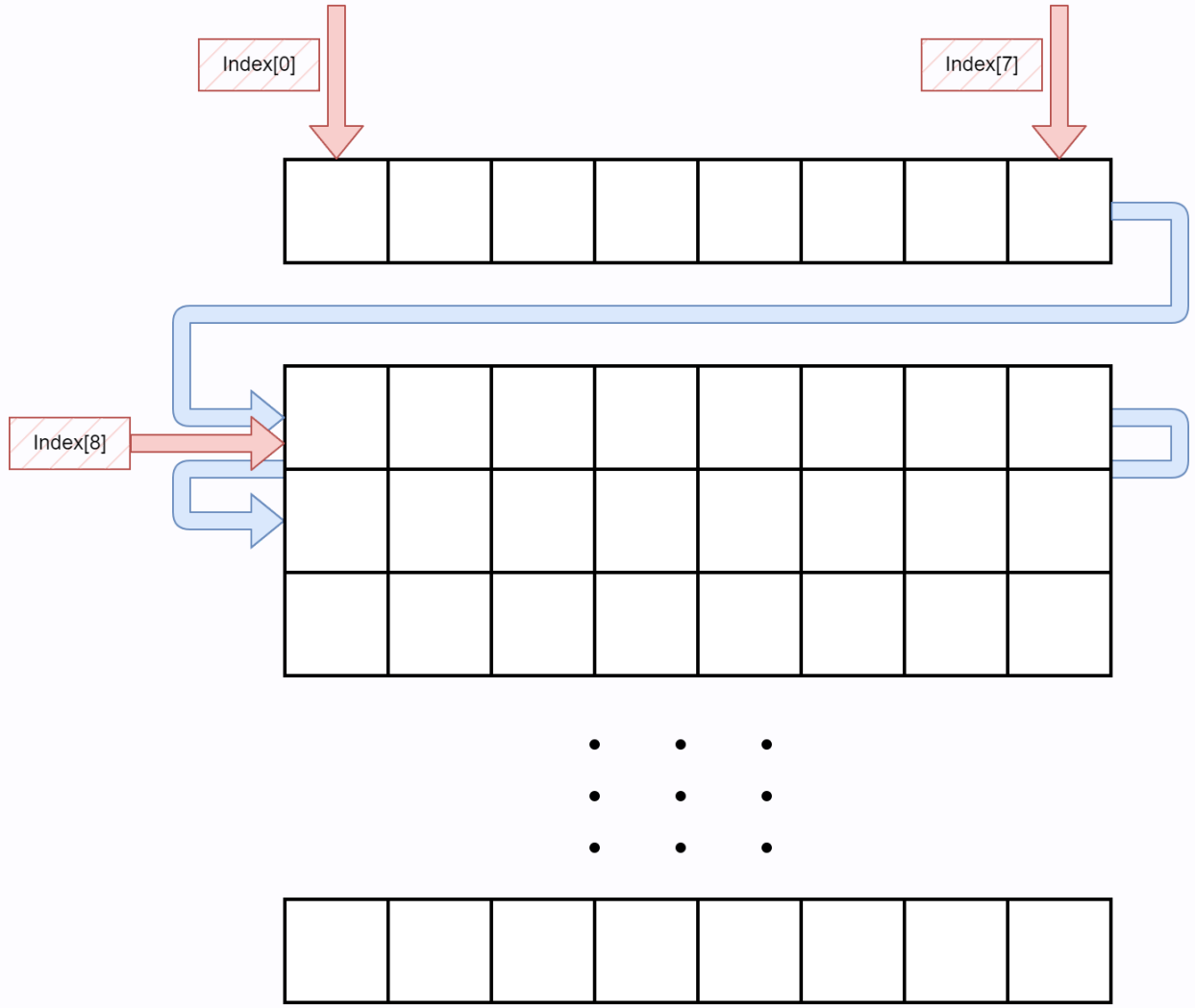
- **Data** is a **physical thing!**
- Its physically **stored somewhere!**
  - Charges in DRAM
  - Charges in CPU Cache
  - Magnetic fields on your hard drive
  - Charges in NAND cells on your SSD
  - *etc...*

How does the computer see data?

- *Very simplified:* Data is stored in **memory**
  - *What memory it is, does not matter*

How does a computer store data?

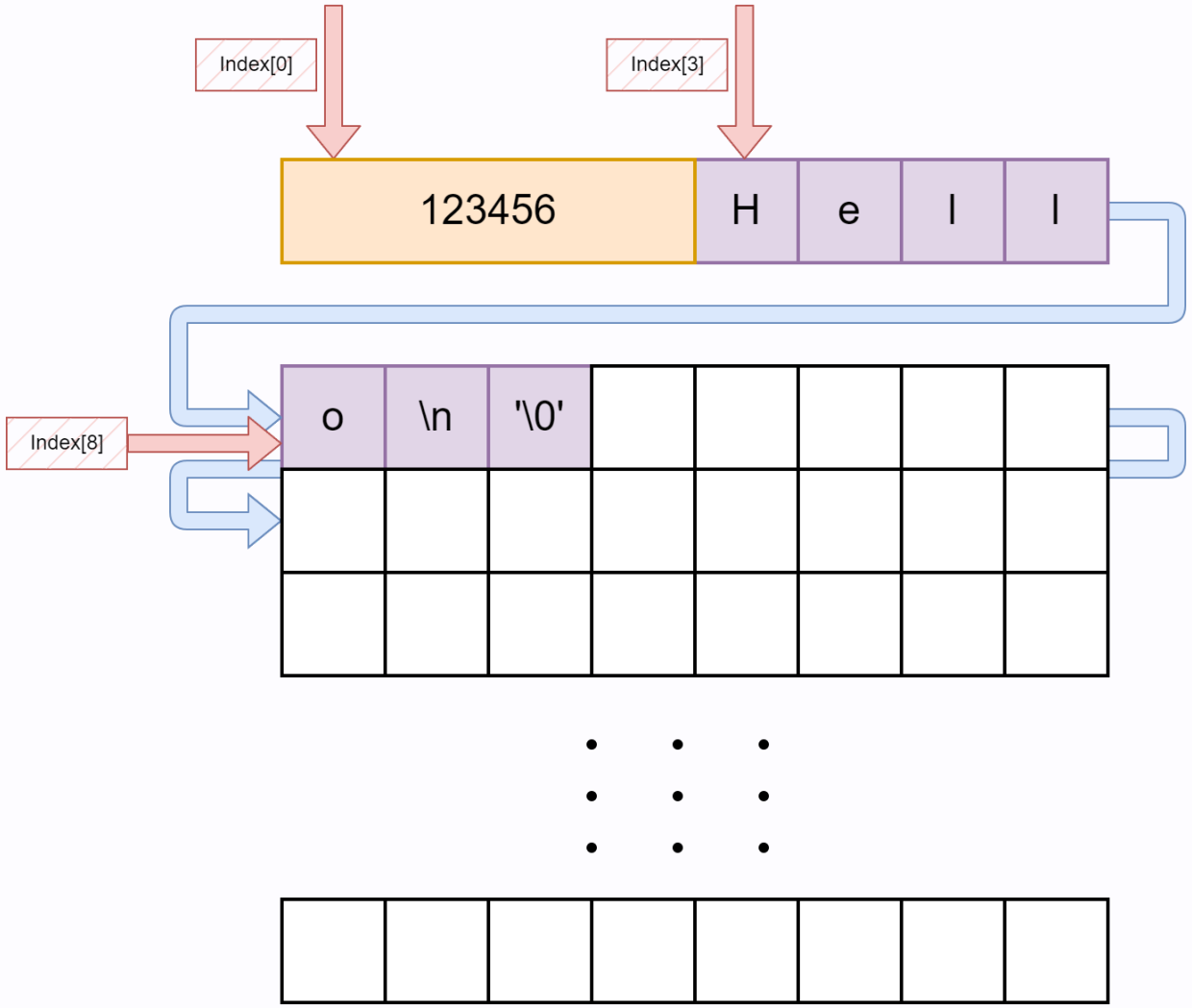
- This **memory** can be though of as a **very long array**
- Each cell contains **1 byte**



How does a computer store data?

- *Note: this is a very simplified example!*

```c
int main(void)
{
    int number = 123456;
    char text[] = "Hello\n";

    return 0;
}
```



Conclusion: *when* we create a variable, it is stored in memory

- Important word: **WHEN**

Example: for loop

Example in `examples/pointers-memory/for-example.c`

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 5; i++)
    {
        printf("%d ", i);
    }
    return 0;
}
// Output: 0 1 2 3 4
```

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 5; i++)
    {
        printf("%d ", i);
    }
    printf("%d ", i);
    return 0;
}
// Output: for-example.c:12:19: error: 'i' undeclared
//                              (first use in this function)
//          12 |     printf("%d ", i);
```

- Why does this not work?

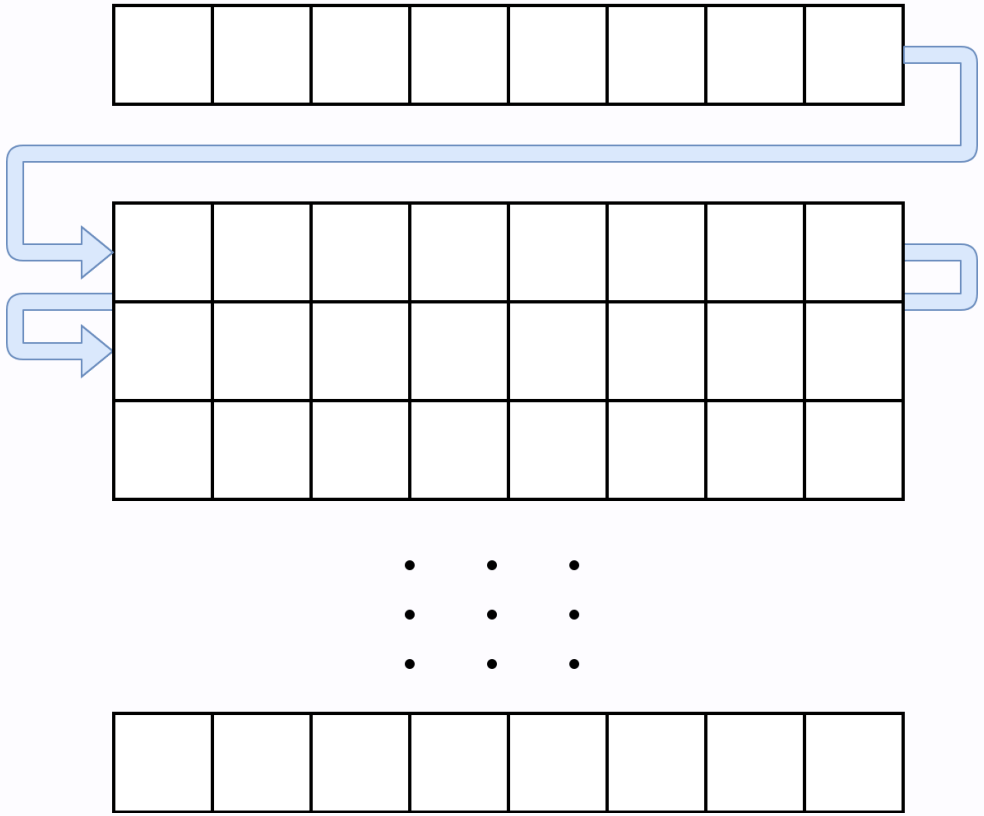
Lets analyse the for example

```c
#include <stdio.h>

int main(void)
{
    // < Function start >
    for (int i = 0; i < 5; i++)
    {
        // < Middle of loop >
        printf("%d ", i);
    }
    // < Function end >
    printf("%d ", i);
    return 0;
}
```

For example: Function start

- At first, no memory is used!

```c
#include <stdio.h>

int main(void)
{
    // < Function start >
    for (int i = 0; i < 5; i++)
    {
        // < Middle of loop >
        printf("%d ", i);
    }
    // < Function end >
    printf("%d ", i);
    return 0;
}
```

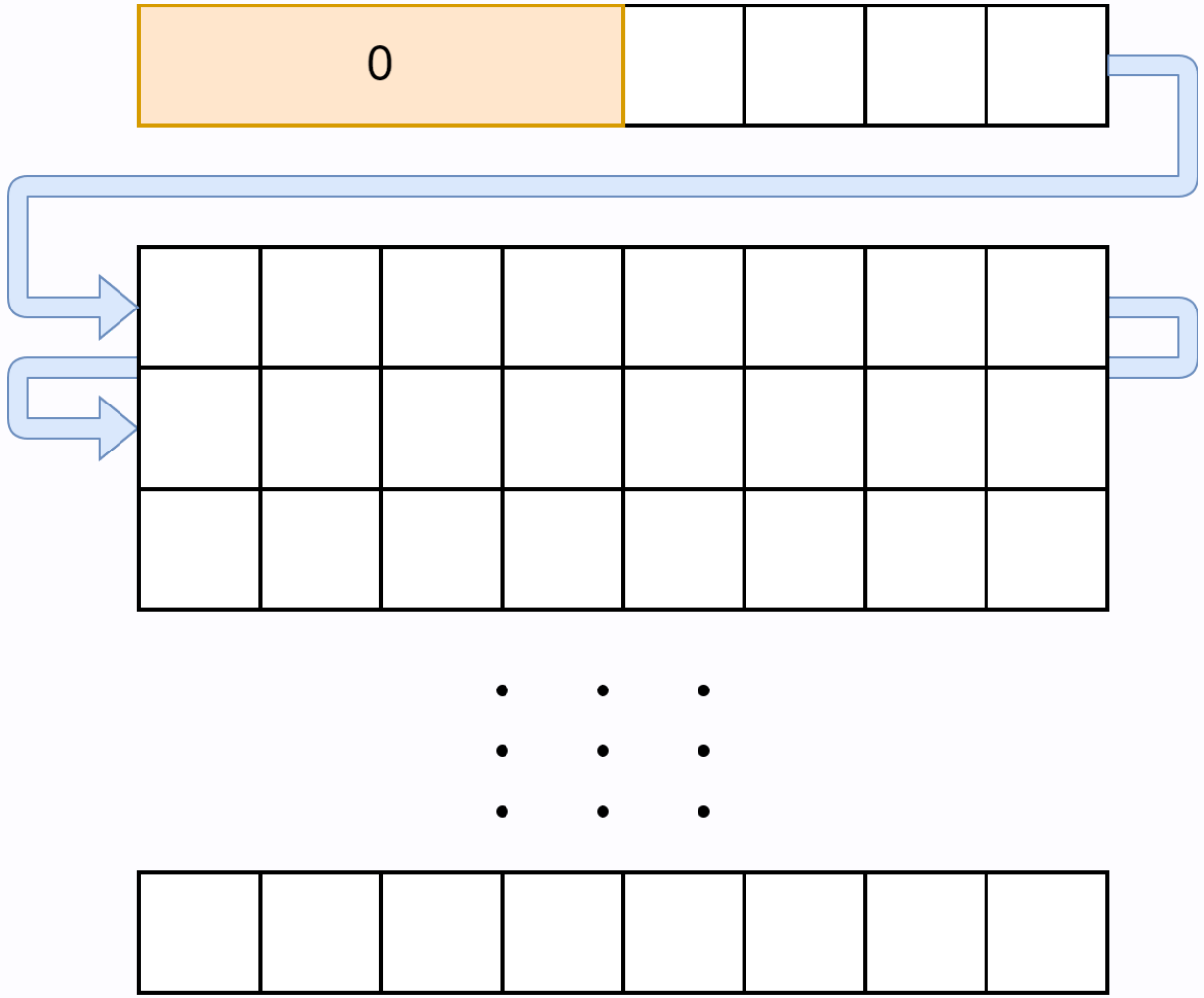


For example:

Middle of loop

- Now we have memory used to store `i`

```c
#include <stdio.h>

int main(void)
{
    // < Function start >
    for (int i = 0; i < 5; i++)
    {
        // < Middle of loop >
        printf("%d ", i);
    }
    // < Function end >
    printf("%d ", i);
    return 0;
}
```

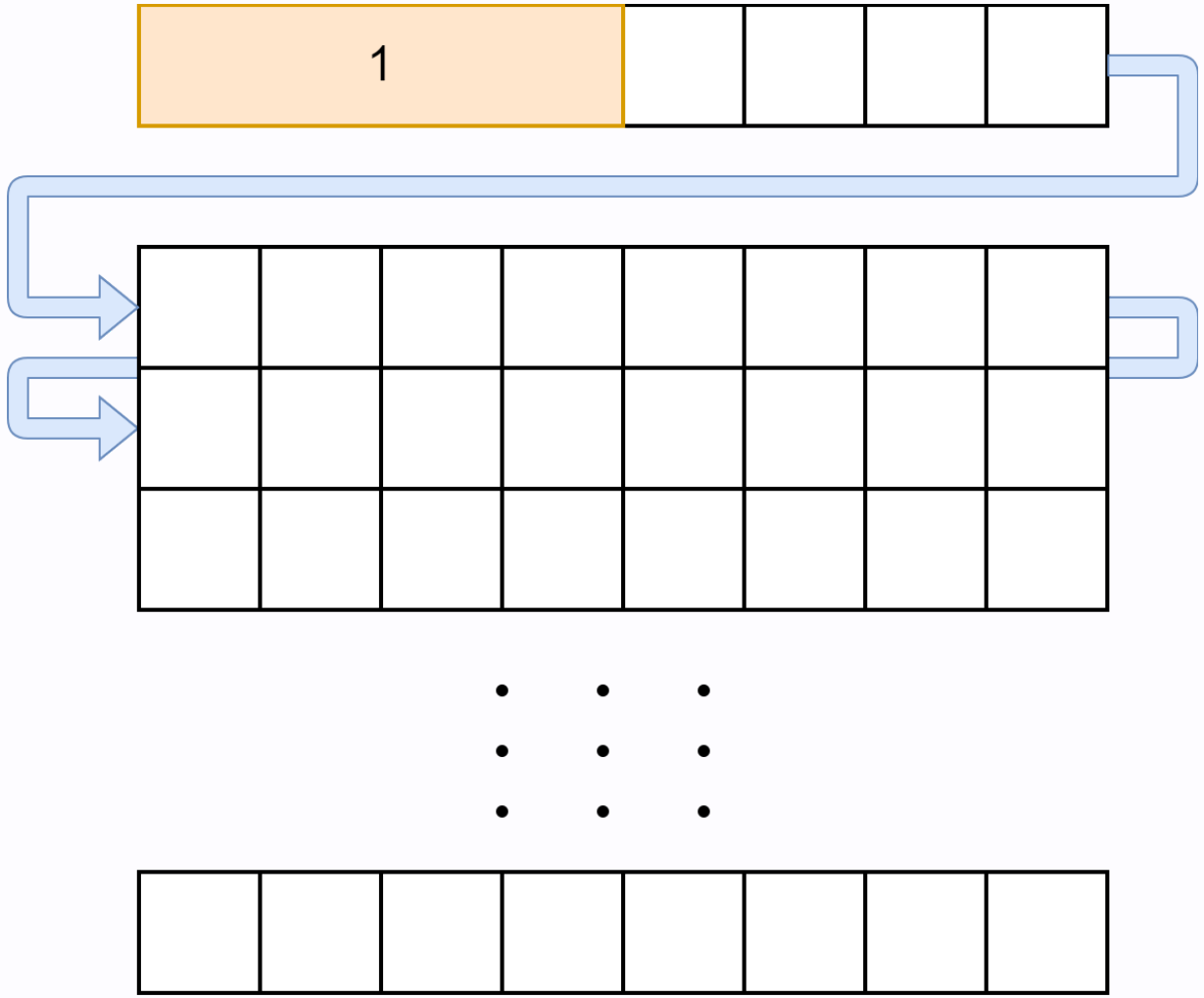


For example:

Middle of loop

- As `i` increases, we increment the value

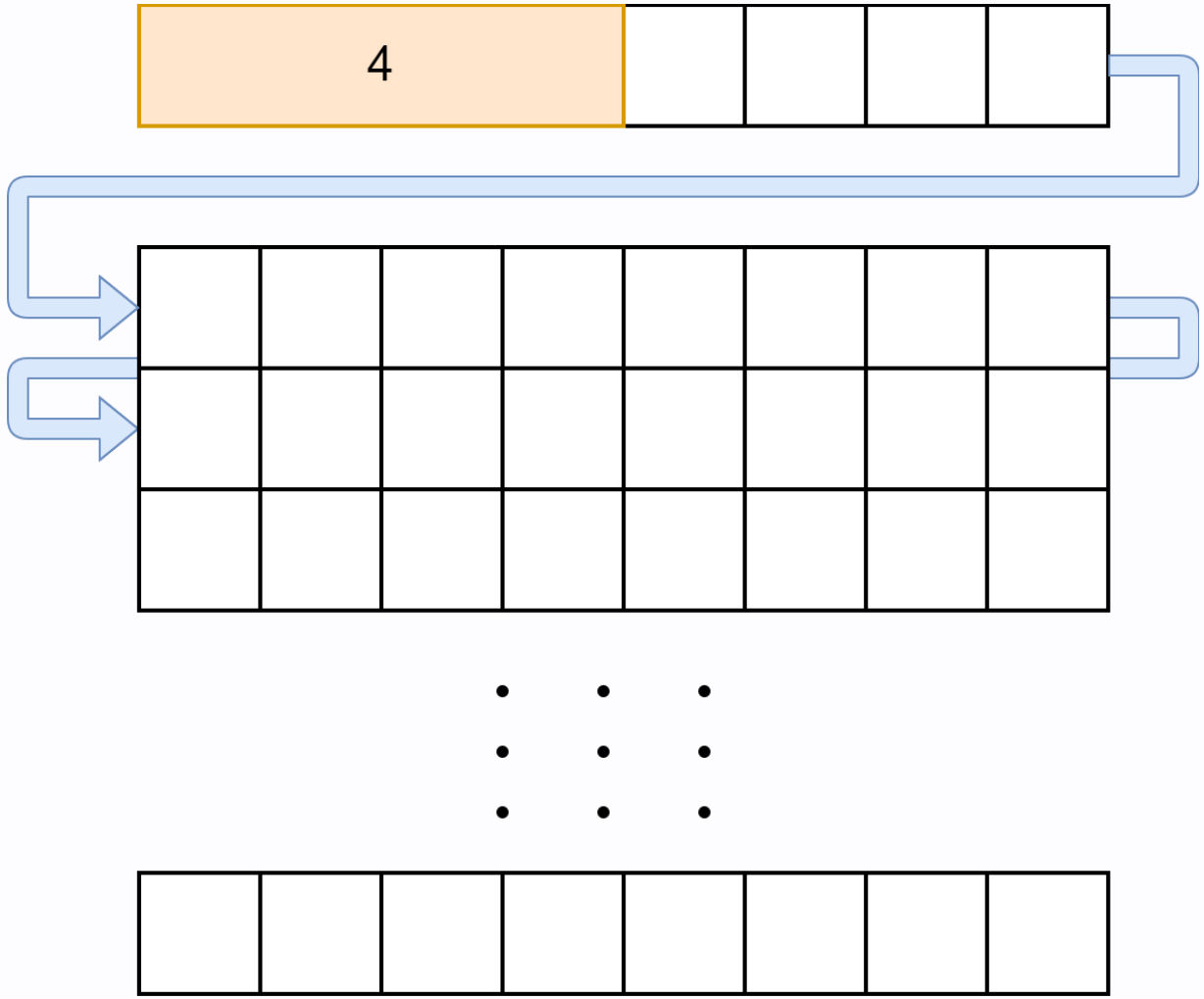
```c
#include <stdio.h>

int main(void)
{
    // < Function start >
    for (int i = 0; i < 5; i++)
    {
        // < Middle of loop >
        printf("%d ", i);
    }
    // < Function end >
    printf("%d ", i);
    return 0;
}
```



For example: Ending of loop

- `i` reached the **maximum value**

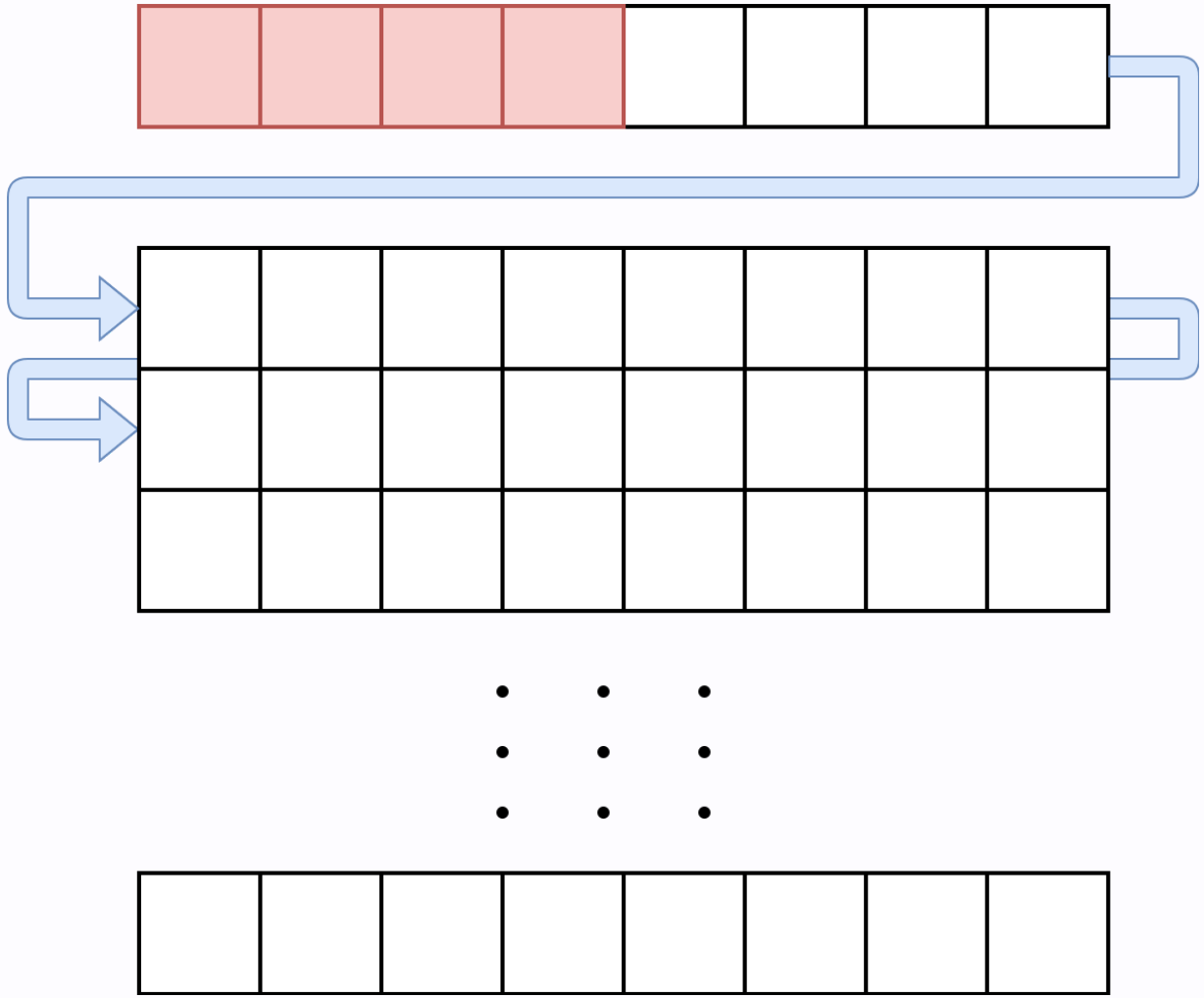
```c
#include <stdio.h>

int main(void)
{
    // < Function start >
    for (int i = 0; i < 5; i++)
    {
        // < Middle of loop >
        printf("%d ", i);
    }
    // < Function end >
    printf("%d ", i);
    return 0;
}
```



For example: Ending of loop

- When we exit the loop `i` value gets **cleared**

```c
#include <stdio.h>

int main(void)
{
    // < Function start >
    for (int i = 0; i < 5; i++)
    {
        // < Middle of loop >
        printf("%d ", i);
    }
    // < Function end >
    printf("%d ", i);
    return 0;
}
```



Conclusion: variables exist only in a certain area of the program

- **Variables** have a certain **lifetime**
  - *How long they can be accessed*
- **Variables** can exist in a certain **scope**
  - *From where they are reachable*

Variable visibility: Block scope

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 5; i++)
    {
        // "i" exists here!
        printf("%d ", i);
    }
    // "i" is OUT OF SCOPE HERE, thus we can't use it!
    printf("%d ", i);
    return 0;
}
// Output: for-example.c:12:19: error: 'i' undeclared
//                              (first use in this function)
//          12 |     printf("%d ", i);
```

Variable visibility: Function scope

```c
#include <stdio.h>

int Multiply(int multiplicant, int multiplier);

int main(void)
{
    int firstMember, secondMember;
    printf("Enter multiplicant: ");
    scanf("%d", &firstMember);

    printf("Enter multiplier: ");
    scanf("%d", &secondMember);

    int result = Multiply(firstMember, secondMember);
    printf("Result is %d!\n", result);
    return 0;
}

int Multiply(int multiplicant, int multiplier)
{
    int result = multiplicant * multiplier;
    // firstMember = 0; // Can't do this here,
    // because it does NOT exist in this function!
    return result;
}
```

Variable visibility: File scope

```c
// examples/pointers-memory/for-global.c
#include <stdio.h>

int i = 0; // Global variable

void Print();

int main(void)
{
    Print();
    for(i = 0; i < 5; i++)
    {
        printf("%d ", i);
    }
    printf("\n");
    Print();
    return 0;
}

void Print()
{
    // i is accesible here!
    i++;
    printf("i is %d\n", i);
}
// Output:  i is 1
//          0 1 2 3 4
//          i is 6
```

- We can create **global variables**
  - *They are accesible anywhere **in the same file!***
- Can be accesed in **other files** if we add the `extern` keyword.

```c
// In another .c file
extern int i;
```

Notes on global variables

- Generaly, they should be **avoided**
  - Limit our *variable naming*
  - Making things more confusing
  - Can introduce undesired side effects
- Mostly used if we need to comunicate events **between functions**

Global variable usage example

```c
#include <stdio.h>
#include <stdint.h>

void ButtonPressedEvent();

// Global variable for buttonState
uint8_t buttonState = 0;

int main(void)
{
    // Infinite loop, so our program runs forever!
    while (1)
    {
        // Our program is waiting for the button...
        if(buttonState)
        {
            printf("Button has been pressed!\n");
            buttonState = 0;
        }
    }
    return 0;
}

/**
 * @brief Called by hardware INTERRUPT when a button is pressed!
 *
 */
void ButtonPressedEvent()
{
    buttonState = 1;
}
```

Coming back to pointers

Why do we need them?

Why do we need them?

```c
// examples/passing-structures/fillStudent.c
#include <stdio.h>
#include <string.h>

#define NAME_LEN 100U

typedef struct
{
    char name[NAME_LEN];
    unsigned int age;
} Person;
```

```c
int main(void)
int main(void)
{
    Person student;
    strncpy(student.name, "Bob", NAME_LEN);
    student.age = 20;
    printf("BeforeMain: Name: %s, age: %d\n", student.name, student.age);

    FillStudentData(student);

    printf("After: Name: %s, age: %d\n", student.name, student.age);
    return 0;
}

int FillStudentData(Person ctx)
{
    printf("BeforeFill: Name: %s, age: %d\n", ctx.name, ctx.age);
    strncpy(ctx.name, "Alice", NAME_LEN);
    ctx.age = 30;
    printf("AfterFill: Name: %s, age: %d\n", ctx.name, ctx.age);
}
```

```c
// Output: BeforeMain: Name: Bob, age: 20
//         BeforeFill: Name: Bob, age: 20
//         AfterFill: Name: Alice, age: 30
//         After: Name: Bob, age: 20
```

We can return the structure!

```c
// examples/passing-structures/fillStudent-return.c
Person FillStudentData(Person ctx);

int main(void)
{
    Person student;
    strncpy(student.name, "Bob", NAME_LEN);
    student.age = 20;
    printf("BeforeMain: Name: %s, age: %d\n", student.name, student.age);

    student = FillStudentData(student);

    printf("After: Name: %s, age: %d\n", student.name, student.age);
    return 0;
}

Person FillStudentData(Person ctx)
{
    printf("BeforeFill: Name: %s, age: %d\n", ctx.name, ctx.age);
    strncpy(ctx.name, "Alice", NAME_LEN);
    ctx.age = 30;
    printf("AfterFill: Name: %s, age: %d\n", ctx.name, ctx.age);
    return ctx;
}
// Output:  BeforeMain: Name: Bob, age: 20
//          BeforeFill: Name: Bob, age: 20
//          AfterFill: Name: Alice, age: 30
//          After: Name: Alice, age: 30
```

Problem solved! Right?

How to know if a function did NOT fail?

Should we care if the function failed?

When to care

- **User input**
  - Users **will break** your system in the most **spectacular ways**
  - It will happen **at the most inconvenient times**
- Interfacing with **hardware**
  - Can be **busy** doing something else
  - Can be **offline**, due to a **low power state**, etc..
  - Can be **broken**

Usualy, we the function *return value* tells us that!

- *But we have used that up...*

Function validation return status

```c
// examples/passing-structures/fillStudent-return-validation.c
// #define DEBUG_TEST_INVALID // Leave this!

int FillStudentData(Person ctx)
{
#ifndef DEBUG_TEST_INVALID
    int fillAge = 30;
#else
    int fillAge = -30;
#endif

    printf("BeforeFill: Name: %s, age: %d\n", ctx.name, ctx.age);
    strncpy(ctx.name, "Alice", NAME_LEN);
    if(fillAge < 0)
    {
        printf("Error: Invalid age!\n");
        return FILL_STUDENT_FAILED;
    }
    ctx.age = fillAge;
    printf("AfterFill: Name: %s, age: %d\n", ctx.name, ctx.age);

    return FILL_STUDENT_OK;
}
```

```c
int main(void)
{
    Person student;
    strncpy(student.name, "Bob", NAME_LEN);
    student.age = 20;
    printf("BeforeMain: Name: %s, age: %d\n", student.name, student.age);

    int status = FillStudentData(student);
    if(status == FILL_STUDENT_OK)
    {
        printf("Function completed succesfully!\n");
    }
    else
        printf("Function failled!\n");

    printf("After: Name: %s, age: %d\n", student.name, student.age);
    return 0;
}
// Output:  BeforeMain: Name: Bob, age: 20
//          BeforeFill: Name: Bob, age: 20
//          AfterFill: Name: Alice, age: 30
//          Function completed succesfully!
//          After: Name: Bob, age: 20
```

Function validation return status

```c
// examples/passing-structures/fillStudent-return-validation.c
#define DEBUG_TEST_INVALID // Uncomment this!

int FillStudentData(Person ctx)
{
#ifndef DEBUG_TEST_INVALID
    int fillAge = 30;
#else
    int fillAge = -30;
#endif

    printf("BeforeFill: Name: %s, age: %d\n", ctx.name, ctx.age);
    strncpy(ctx.name, "Alice", NAME_LEN);
    if(fillAge < 0)
    {
        printf("Error: Invalid age!\n");
        return FILL_STUDENT_FAILED;
    }
    ctx.age = fillAge;
    printf("AfterFill: Name: %s, age: %d\n", ctx.name, ctx.age);

    return FILL_STUDENT_OK;
}
```

```c
int main(void)
{
    Person student;
    strncpy(student.name, "Bob", NAME_LEN);
    student.age = 20;
    printf("BeforeMain: Name: %s, age: %d\n", student.name, student.age);

    int status = FillStudentData(student);
    if(status == FILL_STUDENT_OK)
    {
        printf("Function completed succesfully!\n");
    }
    else
        printf("Function failled!\n");

    printf("After: Name: %s, age: %d\n", student.name, student.age);
    return 0;
}
// Output:  BeforeMain: Name: Bob, age: 20
//          BeforeFill: Name: Bob, age: 20
//          Error: Invalid age!
//          Function failled!
//          After: Name: Bob, age: 20
```

We are at the same place...

Can we do anything?

- Yes! **Pass the address?!**

How does that look?

```c
// examples/passing-structures/fillStudent-return-validation-pointers.c

// Function accepts a POINTER (*)
int FillStudentData(Person *ctx)
{
#ifndef DEBUG_TEST_INVALID
    int fillAge = 30;
#else
    int fillAge = -30;
#endif

    // We are accessing a POINTER, thus we use (->)
    printf("BeforeFill: Name: %s, age: %d\n", ctx->name, ctx->age);
    if(fillAge < 0)
    {
        printf("Error: Invalid age!\n");
        return FILL_STUDENT_FAILED;
    }
    ctx->age = fillAge;
    strncpy(ctx->name, "Alice", NAME_LEN);
    printf("AfterFill: Name: %s, age: %d\n", ctx->name, ctx->age);

    return FILL_STUDENT_OK;
}
```

```c
#define DEBUG_TEST_INVALID

int FillStudentData(Person *ctx);

int main(void)
{
    Person student;
    strncpy(student.name, "Bob", NAME_LEN);
    student.age = 20;
    printf("BeforeMain: Name: %s, age: %d\n", student.name, student.age);

    // We pass an ADDRESS (& operator)
    int status = FillStudentData(&student);
    if(status == FILL_STUDENT_OK)
    {
        printf("Function completed succesfully!\n");
    }
    else
        printf("Function failled!\n");

    printf("After: Name: %s, age: %d\n", student.name, student.age);
    return 0;
}
// Output: BeforeMain: Name: Bob, age: 20
//         BeforeFill: Name: Bob, age: 20
//         Error: Invalid age!
//         Function failled!
//         After: Name: Bob, age: 20
```

How does that look?

```c
// examples/passing-structures/fillStudent-return-validation-pointers.c

// Function accepts a POINTER (*)
int FillStudentData(Person *ctx)
{
#ifndef DEBUG_TEST_INVALID
    int fillAge = 30;
#else
    int fillAge = -30;
#endif

    // We are accessing a POINTER, thus we use (->)
    printf("BeforeFill: Name: %s, age: %d\n", ctx->name, ctx->age);
    if(fillAge < 0)
    {
        printf("Error: Invalid age!\n");
        return FILL_STUDENT_FAILED;
    }
    ctx->age = fillAge;
    strncpy(ctx->name, "Alice", NAME_LEN);
    printf("AfterFill: Name: %s, age: %d\n", ctx->name, ctx->age);

    return FILL_STUDENT_OK;
}
```

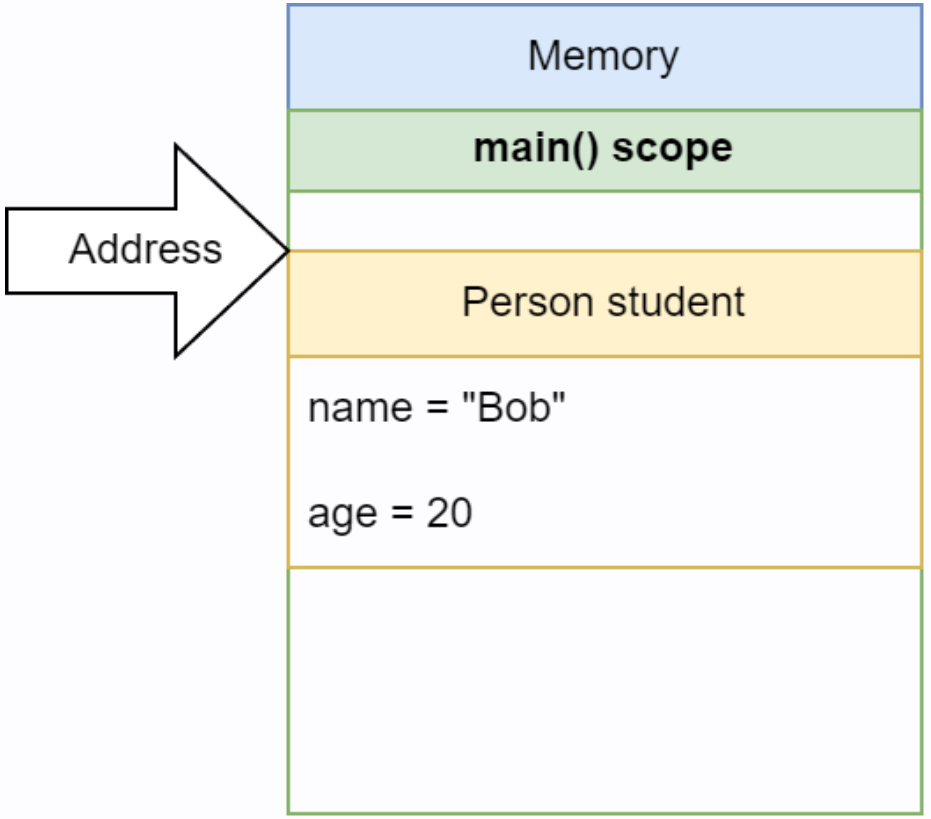
```c
// #define DEBUG_TEST_INVALID

int FillStudentData(Person *ctx);

int main(void)
{
    Person student;
    strncpy(student.name, "Bob", NAME_LEN);
    student.age = 20;
    printf("BeforeMain: Name: %s, age: %d\n", student.name, student.age);

    // We pass an ADDRESS (& operator)
    int status = FillStudentData(&student);
    if(status == FILL_STUDENT_OK)
    {
        printf("Function completed succesfully!\n");
    }
    else
        printf("Function failled!\n");

    printf("After: Name: %s, age: %d\n", student.name, student.age);
    return 0;
}
// Output: BeforeMain: Name: Bob, age: 20
//         BeforeFill: Name: Bob, age: 20
//         AfterFill: Name: Alice, age: 30
//         Function completed succesfully!
//         After: Name: Alice, age: 30
```

Why does it work?

Program start

```
Person student;
strncpy(student.name, "Bob", NAME_LEN);
student.age = 20;
```

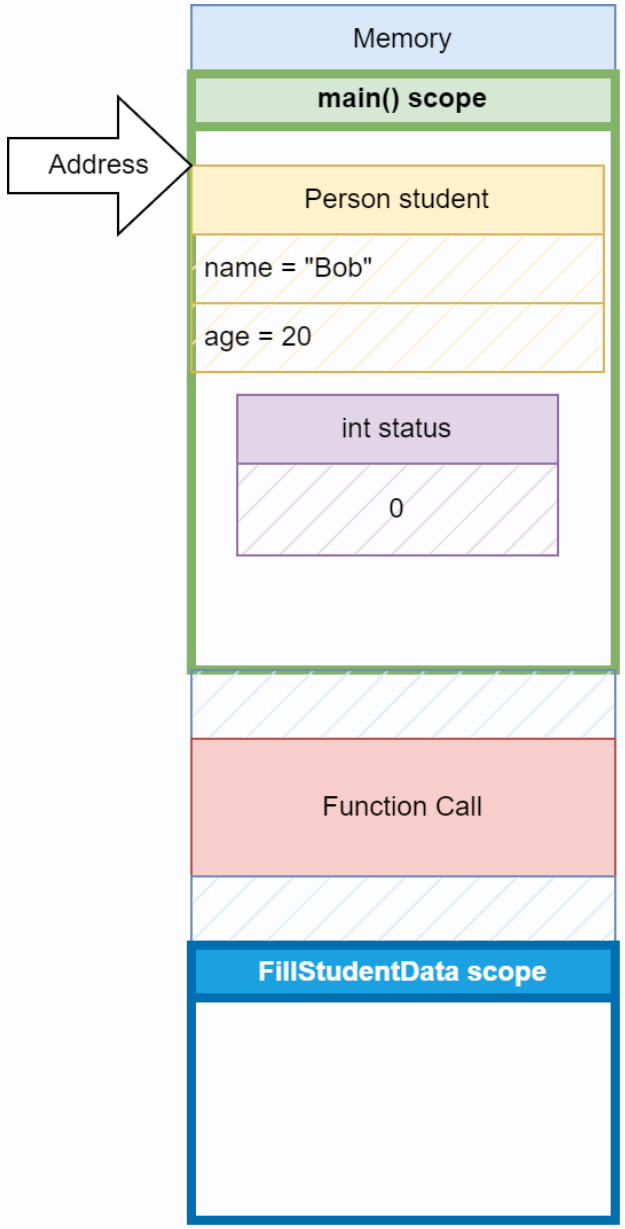
- Data is alocated and written in the **main scope**



Program calling FillStudentData

```
// We pass an ADDRESS (& operator)
int status = FillStudentData(&student);
```

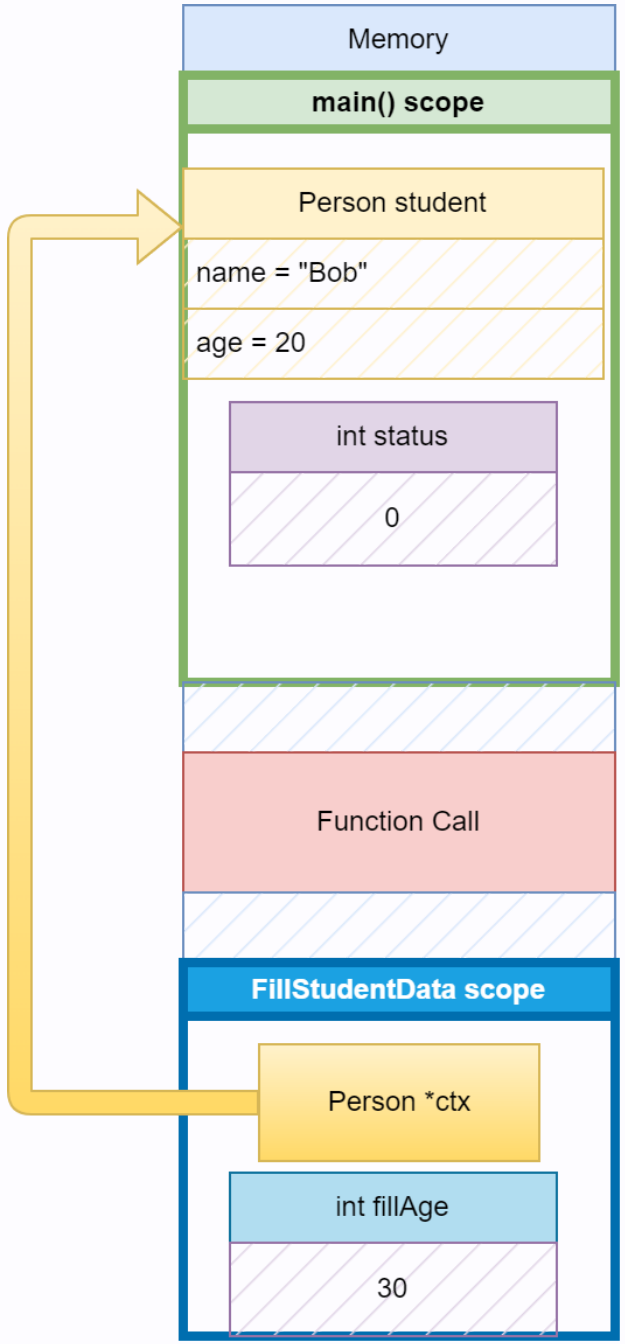
- A function call is made
- A **new scope** ***FillStudentData*** is created!



Using the person structure

```c
// examples/passing-structures/fillStudent-return-validation-pointers.c

int FillStudentData(Person *ctx)
{
#ifndef DEBUG_TEST_INVALID
    int fillAge = 30;
#else
    int fillAge = -30;
#endif

    // We are accessing a POINTER, thus we use (->)
    printf("BeforeFill: Name: %s, age: %d\n", ctx->name, ctx->age);
```

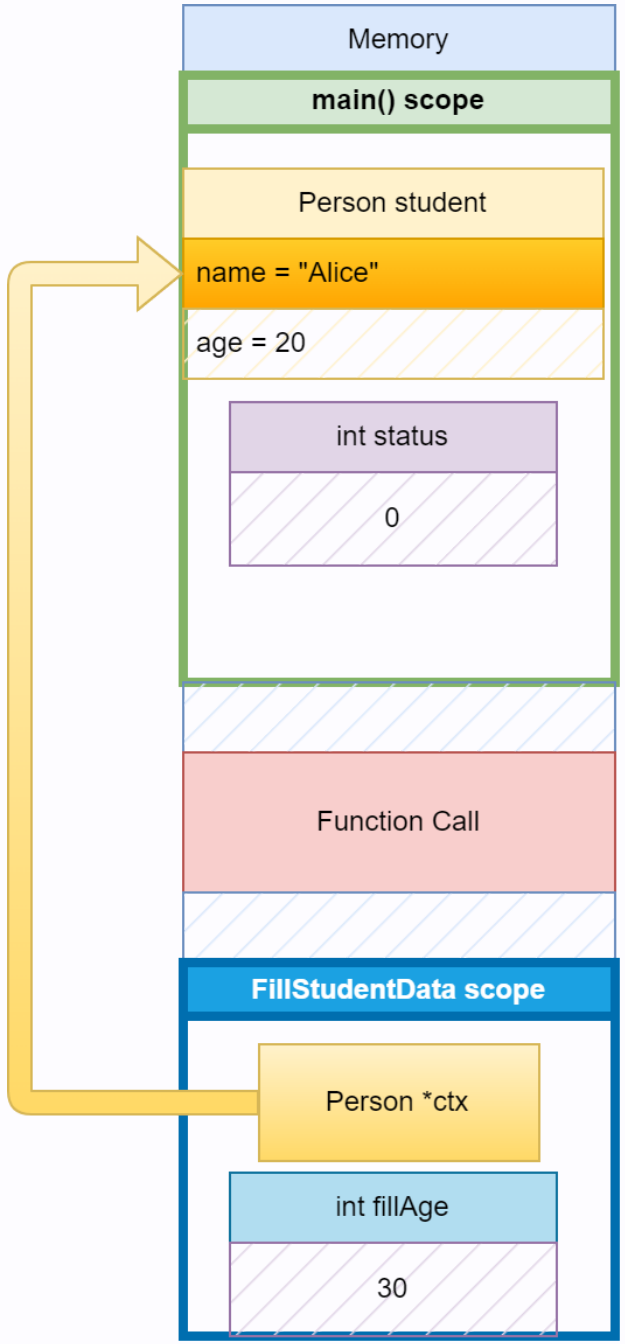
- Unlike normally passed values, pointers **do not copy any data over!**



Using the person structure

```
    if(fillAge < 0)
    {
        printf("Error: Invalid age!\n");
        return FILL_STUDENT_FAILED;
    }
    strncpy(ctx->name, "Alice", NAME_LEN);
    ctx->age = fillAge;
    printf("AfterFill: Name: %s, age: %d\n", ctx->name, ctx->age);

    return FILL_STUDENT_OK;
}
```

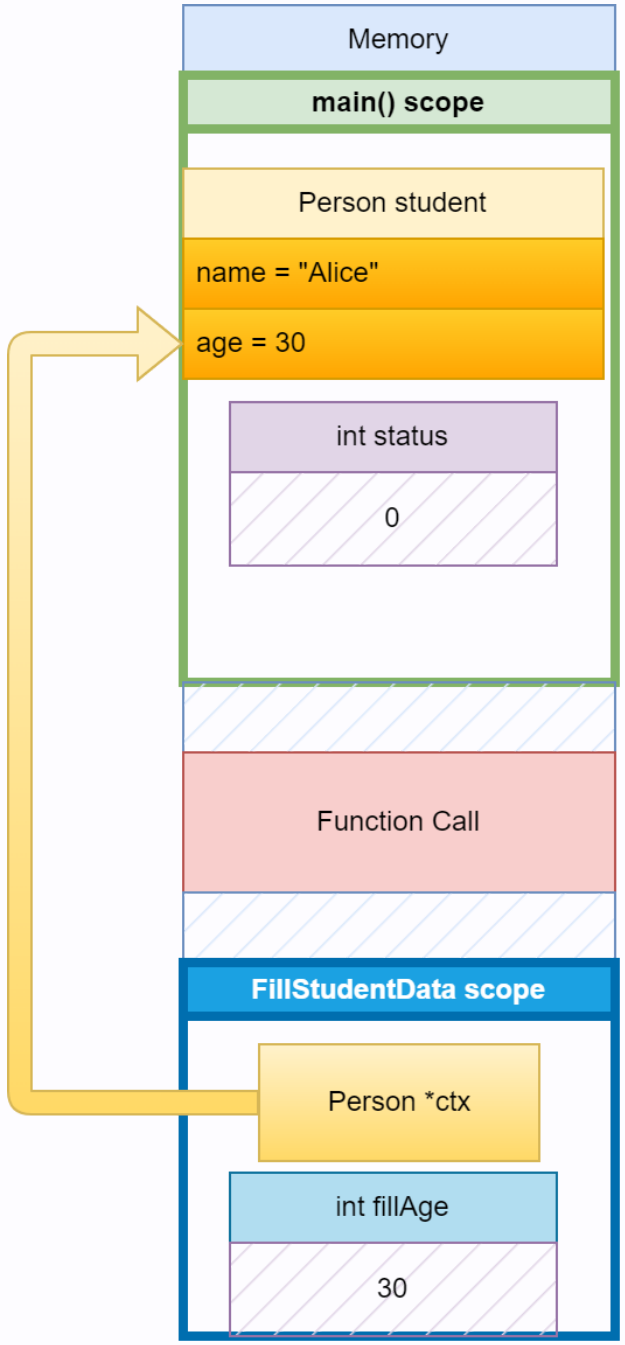
- We are modifying existing values **in their place!**
  - In this case the **main() scope**



Using the person structure

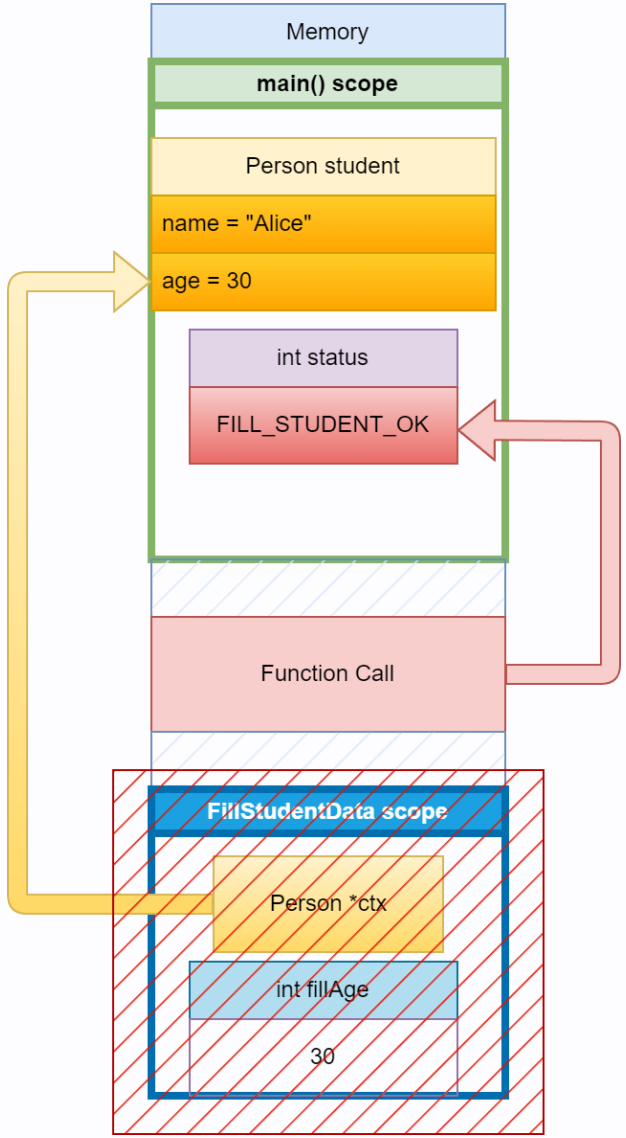
```
    if(fillAge < 0)
    {
        printf("Error: Invalid age!\n");
        return FILL_STUDENT_FAILED;
    }
    strncpy(ctx->name, "Alice", NAME_LEN);
    ctx->age = fillAge;
    printf("AfterFill: Name: %s, age: %d\n", ctx->name, ctx->age);

    return FILL_STUDENT_OK;
}
```

- We are modifying existing values **in their place!**
- In this case the **main() scope**



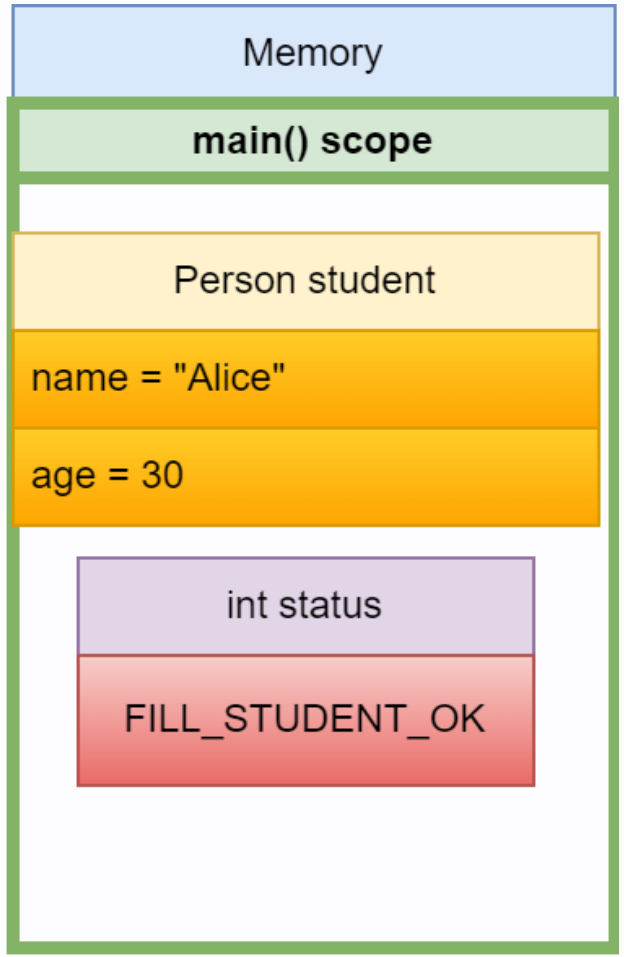
Function return

- The *FillStudentStatus* scope is **cleared**.
- Return value is **saved**



Return to main

- We return back to **main**
- Our variables have been **modified**



Now the hard part is over,

debugging!

Compiler directives with `#define`

- Change the way the program compiles with a `#define`
- `#ifdef` - checks if a specified **define exists**
- Block must end with `#endif` !

```c
#define TEST_DEFINE

// This will be included in the source code!
#ifdef TEST_DEFINE
    printf("Test is defined!\n");
#endif

// This WON'T be included!
#ifndef TEST_DEFINE
    printf("Test is not defined!\n");
#endif
```

Example: Header guards

- To protect us from including a header file **twice**, we add *header guards*.

```
#ifndef LOGGING_H // Name of your header
#define LOGGING_H

/* Header code... */

#endif
```

- *On modern compilers* `#pragma once` *can also be used*

Example: Togling debug output

```
#ifdef A9G_UART_DMA_DEBUG_OUTPUT
    my_printf("A9G_Device.tranmitState: %d\n", A9G_Device.tranmitState);
    my_printf("A9G_Device.RxState: %d\n", A9G_Device.phGPS->RxState);
    my_printf("A9G_Device.ErrorCode: %d\n", A9G_Device.phGPS->ErrorCode);
#endif

    if (A9G_Device.tranmitState & GPS_LISTEN_IT)
        ret = HAL_UART_DMAResume(A9G_Device.phGPS);
    else
        ret = HAL_UART_Receive_DMA(A9G_Device.phGPS, buffer, size * 2);

    if (ret != HAL_OK)
    {
        LogError("UART_Receive_DMA failled!", "at A9G.c:34");
        A9G_Device.tranmitState &= ~GPS_LISTEN_DMA_SINGLE;
        A9G_Device.tranmitState &= ~GPS_LISTEN_DMA;
#ifdef A9G_UART_DMA_DEBUG_OUTPUT
        my_printf("AFTER: A9G_Device.tranmitState: %d\n", A9G_Device.tranmitState);
        my_printf("AFTER: A9G_Device.RxState: %d\n", A9G_Device.phGPS->RxState);
        my_printf("AFTER: A9G_Device.ErrorCode: %d\n", A9G_Device.phGPS->ErrorCode);
#endif
        return A9G_DMA_ERROR;
    }
    return A9G_OK;
```

`#define` macros

- Code snippet substitution.

```
#define MACRO_NAME(inputs)((operation))
```

```
#define SUM(one, two)((one + two))

int result = SUM(10, 2);
/* This will get compiled to:
  int result = 10 + 2;
*/
```

Putting it together

```
extern A9G_reciever_t A9G_Device; // This means the structure is defined somewhere else.

uint8_t A9G_Wakeup()
{
    A9G_message resp = {0};

    if(A9G_Device.powerState == A9G_RUN)
        return 0;

    A9G_Device.enbleSleepEntry = 0;
    TIM2_TimeoutDuration(TIM2_1S_TIMEOUT);
    HAL_GPIO_WritePin(A9G_LOW_POWER_ENTRY_GPIO_Port, A9G_LOW_POWER_ENTRY_Pin, 0);
    HAL_Delay(10);


    if(AT_Command_send(A9G_EXIT_LOW_POWER, AT_SINGLE_CAPTURE, &resp) != A9G_OK)
    {
        LogError("Failed to wakeup A9G from sleep!", "at A9G_user_functions.c:97");
        return 1;
    }


    if(AT_Command_send(AT_TEST, AT_SINGLE_CAPTURE, &resp) != A9G_OK)
    {
        LogError("Failed to wakeup A9G from sleep!", "at A9G_user_functions.c:104");
        TIM2_TimeoutDuration(TIM2_10S_TIMEOUT);
        return 1;
    }

    if(resp.responceCode != A9G_RESP_OK)
    {
        A9G_Device.enbleSleepEntry = 1;
        TIM2_TimeoutDuration(TIM2_10S_TIMEOUT);
        return 1;
    }

    LogEvent("A9G Woke up from SLEEP!", "at A9G_user_functions.c:118");
    A9G_Device.powerState = A9G_RUN;
    return 0;
}
```

A9G_reciever structure

```
typedef struct
{
    A9G_POWER_STATE powerState;
    UART_HandleTypeDef *phGPS;
    volatile A9G_UART_STATUS_t tranmitState;
    UART_HandleTypeDef *phGSM;
    uint8_t buff[A9G_BUFF_SIZE];
    uint16_t sendTimeout;
    A9G_GPS_STATUS_t gpsStatus;
    A9G_GSM_STATUS_t smsStatus;
    uint8_t enableCallbacks;
    uint8_t enbleSleepEntry;
    uint8_t availableSms;
    A9G_HTTP_t http;
} A9G_reciever_t;
```