Arrays, Pointers and Variable Scope

MATH 5061: Fundamentals of Computer Programming for Scientists and Engineers

Dr. Richard Berger
richard.berger@temple.edu

Department of Mathematics
Temple University

10/25/2016

TEMPLE UNIVERSITY

Outline

Outline

Arrays

Declaration:

```
T  a [ 100 ] ;
```

- an array is a data type representing a *fixed* size, *consecutive* sequence of elements
- in an array of type `T`, each element has the data type `T`
- the total amount of memory needed to store an array therefore is `length * sizeof(T)`

Arrays

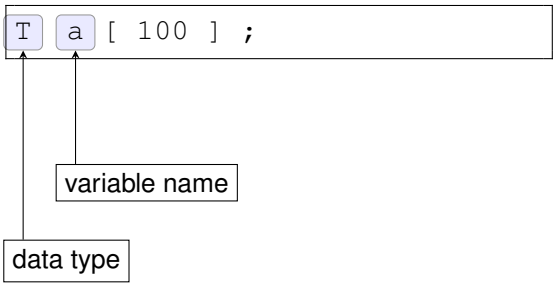
Declaration:

```
T  a [ 100 ] ;
```

- an array is a data type representing a *fixed* size, *consecutive* sequence of elements
- in an array of type `T`, each element has the data type `T`
- the total amount of memory needed to store an array therefore is `length * sizeof(T)`

Arrays

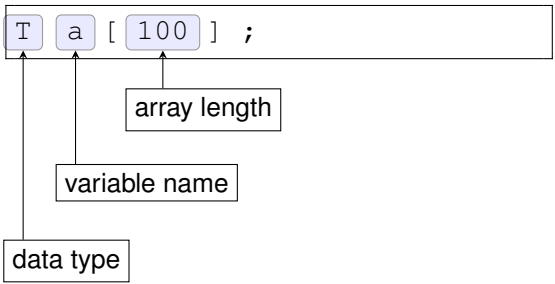
Declaration:



- an array is a data type representing a *fixed* size, *consecutive* sequence of elements
- in an array of type `T`, each element has the data type `T`
- the total amount of memory needed to store an array therefore is `length * sizeof(T)`

Arrays

Declaration:



- an array is a data type representing a *fixed* size, *consecutive* sequence of elements
- in an array of type `T`, each element has the data type `T`
- the total amount of memory needed to store an array therefore is `length * sizeof(T)`

Arrays

Declaration:

```
T  a [ 100 ] ;
```

Examples:

```
int iarray[100];
double darray[100];

// sizeof(iarray) = 400 bytes
// sizeof(darray) = 800 bytes
```

- an array is a data type representing a *fixed* size, *consecutive* sequence of elements
- in an array of type `T`, each element has the data type `T`
- the total amount of memory needed to store an array therefore is `length * sizeof(T)`

Arrays

Declaration:

```
T  a [ 100 ] ;
```

Examples:

```
int iarray[100];
double darray[100];

// sizeof(iarray) = 400 bytes
// sizeof(darray) = 800 bytes
```

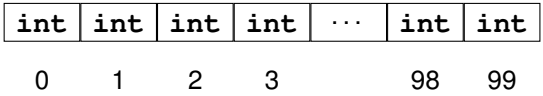
- an array is a data type representing a *fixed* size, *consecutive* sequence of elements
- in an array of type `T`, each element has the data type `T`
- the total amount of memory needed to store an array therefore is `length * sizeof(T)`

Note

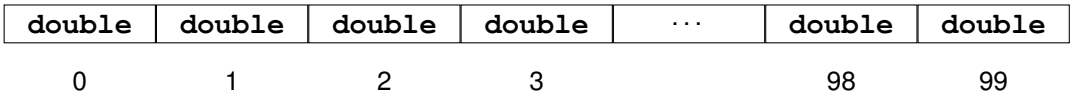
Unlike Python lists, an array in C/C++ can not store a mixture of different types.

Visualization of arrays

int a[100]



double b[100]



- The amount of memory occupied by an array depends on its data type.
- On a system with 32bit (4 byte) integers and 64bit (8 byte) double floating-point numbers the same sequence of numbers will occupy twice as much memory in the floating point format compared to an integer array.

Accessing array elements

Writing a value:

`varname[index] = value;`

```
int a[100];
a[0] = 2;
a[1] = 11;
a[2] = 6;
a[3] = 3;
```

Reading a value: `varname[index]`

```
printf("%d", a[3]);
```

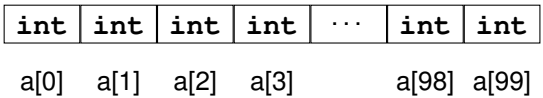
- Elements of an array of size N are accessible with indices from 0 to $N-1$.
- An array variable points to the first element of the sequence. The index then represents the offset needed to get the element you want.
- Index 0 points to the first element, while index 1 points to the second.

Note

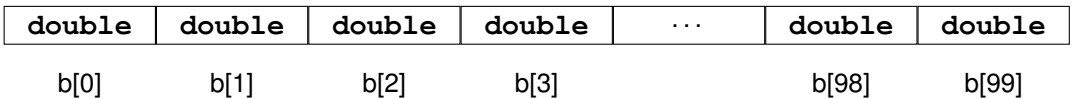
There is no slicing syntax like in Python. In C/C++ you can only access **one single element** using a **positive** index number.

Accessing array elements

int a[100]



double b[100]



- Q: What is the value of each element in an array after it is declared?

```
int a[1000];
a[500] = ?
```

- Q: What is the value of each element in an array after it is declared?

```
int a[1000];
a[500] = ?
```

- **A: We don't know.** It's undefined. But not random.

Live Demo: Example 1

Example 1: show uninitialized values of an array

```
#include <stdio.h>

int main() {
    int a[1000];

    for(int i = 0; i < 1000; ++i) {
        printf("a[%d]: %d\n", i, a[i]);
    }

    return 0;
}
```

Initialization of variables

- $\Rightarrow$ before reading values from variable you should always initialize them
- Declaration of a variable only gives you enough memory so its data fits in.
- It does **not** set a default value!

Initialization of array variables

Manual way:

```
int a[4];
a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
```

Initialization of array variables

Manual way:

```
int a[4];
a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
```

Using initializer syntax:

```
int a[4]{1,2,3,4};
```

even this works:

```
int a[]{1,2,3,4};
```

the compiler infers the size from the initialization

Initialization of array variables

Manual way:

```
int a[4];
a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
```

Using initializer syntax:

```
int a[4]{1,2,3,4};
```

even this works:

```
int a[]{1,2,3,4};
```

the compiler infers the size from the initialization

Zero initialization:

if all elements should be initialized to zero we can do this:

```
int a[4]{0};
```

- Q: If we define an array like this, how do we get the number of elements in the array?

```
int a[] {51, 20, 52, 55, 1, 29, 39, 47, 19, 74, 71, 43,
         82, 38, 17, 80, 69, 72, 1, 67, 68, 19, 1, 24,
         18, 48, 99, 52, 55, 29, 91, 46, 49, 98, 51,
         38, 32, 84, 95, 29, 60, 91, 23, 50, 10, 90,
         12, 18, 65, 27, 86, 56, 94, 92, 17, 81, 13,
         91, 93, 99, 11, 11, 45, 54, 54, 47, 42, 90,
         29, 44, 9, 51, 84, 95, 43, 60, 79, 44, 80,
         4, 9, 53, 44, 94, 45, 18, 85, 88, 55, 94,
         31, 44, 61, 3, 24, 51, 41, 53, 17, 42};
```

- Q: If we define an array like this, how do we get the number of elements in the array?

```
int a[] {51, 20, 52, 55, 1, 29, 39, 47, 19, 74, 71, 43,
         82, 38, 17, 80, 69, 72, 1, 67, 68, 19, 1, 24,
         18, 48, 99, 52, 55, 29, 91, 46, 49, 98, 51,
         38, 32, 84, 95, 29, 60, 91, 23, 50, 10, 90,
         12, 18, 65, 27, 86, 56, 94, 92, 17, 81, 13,
         91, 93, 99, 11, 11, 45, 54, 54, 47, 42, 90,
         29, 44, 9, 51, 84, 95, 43, 60, 79, 44, 80,
         4, 9, 53, 44, 94, 45, 18, 85, 88, 55, 94,
         31, 44, 61, 3, 24, 51, 41, 53, 17, 42};
```

- A: By using the **sizeof** operator:

```
int nelement = sizeof(a) / sizeof(int);
```

Example

```c
#include <stdio.h>

int main() {
  int numbers[] {10, 20, 30, 40, 50};
  int count = sizeof(numbers) / sizeof(int);
  float average = 0.0;

  for (int i = 0; i < count; ++i) {
    average += numbers[i];
  }

  average /= count;

  printf("average: %f\n", average);
}
```

Character strings

```
#include <stdio.h>

int main() {
    char str[] = "Hello";
    const int count = sizeof(str);

    for(int i = 0; i < count; ++i) {
        printf("str[%d]: %c\n", i, str[i]);
    }

    return 0;
}
```

```
str[0]: H
str[1]: e
str[2]: l
str[3]: l
str[4]: o
str[5]:
```

- C-strings are character arrays
- they are terminated by a 0-byte, which is why they occupy $N+1$ characters

Character strings

```
char str[] = "Hello";
```

is the same as:

```
char str[] {'H', 'e', 'l', 'l', 'o', 0};
```

or

```
char str[] {"Hello"};
```

Live Demo: Example 2

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

int main() {
    char buffer[50];
    char firstname[] = "Richard";
    char lastname[]  = "Berger";

    int len1 = strlen(firstname); // =7
    int len2 = strlen(lastname);  // =6
    int offset = 0;

    for(int i = 0; i < len1; ++i)
        buffer[offset++] = firstname[i];

    buffer[offset++] = ' ';

    for(int i = 0; i < len2; ++i)
        buffer[offset++] = lastname[i];

    buffer[offset] = 0; // terminate string

    printf("%s (len: %d)\n", buffer, strlen(buffer));
    return 0;
}
```

Example: using `strcpy` and `strcat`

- the `string.h` header defines utility functions for modifying string arrays

`strcpy(src, dest)`

Copy characters from `src` array to `dest` array and terminate `dest` with 0-byte.

`strcat(src, dest)`

Append characters from `src` to existing string in `dest` and terminate `dest` with 0-byte.

Example: using `strcpy` and `strcat`

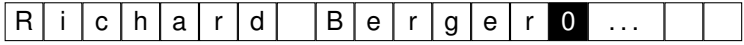
```c
#include <stdio.h>
#include <string.h>

int main() {
    char buffer[50];
    char firstname[] = "Richard";
    char lastname[]  = "Berger";

    strcpy(firstname, buffer);
    strcat(" ", buffer);
    strcat(lastname, buffer);

    printf("%s (len: %d)\n", buffer, strlen(buffer));
    return 0;
}
```

Example: using `strcpy` and `strcat`

1.) `strcpy(firstname, buffer);`

2.) `strcat(" ", buffer);`

3.) `strcat(lastname, buffer);`



Comparing strings

```
int main() {
    char a[] = "Hello World!";
    char b[] = "Hello!";
    bool equal = true;

    for(int i = 0; a[i] && b[i]; ++i) {
        if(a[i] != b[i]) {
            equal = false;
            break;
        }
    }

    if(equal)
        printf("These two are equal!\n");
    else
        printf("These two are NOT equal!\n");

    return 0;
}
```

Comparing strings with `strcmp`

```
#include <stdio.h>
#include <string.h>

int main() {
    char a[] = "Hello World!";
    char b[] = "Hello!";

    if(!strcmp(a, b))
        printf("These two are equal!\n");
    else
        printf("These two are NOT equal!\n");

    return 0;
}
```

- `strcmp(a,b)` returns either a value smaller, equal or larger than 0.
- if `a` alphabetically comes before `b`, a value smaller than 0 is returned
- if `a` and `b` are equal 0 is returned
- if `a` alphabetically comes after `b`, a value larger than 0 is returned
- by checking for `!0 == true` we detect equality

Multi-dimensional Arrays

- Arrays can have more than one dimension by adding more brackets
- Data is arranged sequentially using row-major order

```
double matrix[rows][columns];
```

```
double matrix[2][3];
```

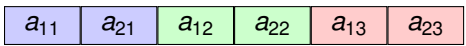
Initialization

```
double matrix[2][3] = { {1, 2, 3},
                        {4, 5, 6} };
```

Memory Layouts of Multi-dimensional Arrays

Column-major order

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

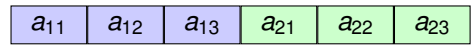


Used by:

- Fortran
- MATLAB
- Julia
- R

Row-major order

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$



Used by:

- C/C++/C#
- Mathematica
- Pascal
- NumPy

Example: Matrix-Vector Multiplication

$$A = \begin{bmatrix} 1 & -1 & 2 \\ 0 & -3 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}$$

```
int A[][] = {{1, -1, 2}, {0, -3, 1}};
int b[]   = {2, 1, 0};
int c[2];

for(int i = 0; i < 2; ++i) {
    c[i] = 0;
    for(int j = 0; j < 3; ++j) {
        c[i] += A[i][j] * b[j];
    }
}
```

Outline

Memory

- Memory on a computer can be seen as one consecutive chunk of data
- Think of it as an *array of bytes*
- Each byte in memory has a unique index, which is its **address**
- the number of bits used to store a memory address determines the amount of memory you can access:

32 bits: up to 2^{32} bytes = 4 GB

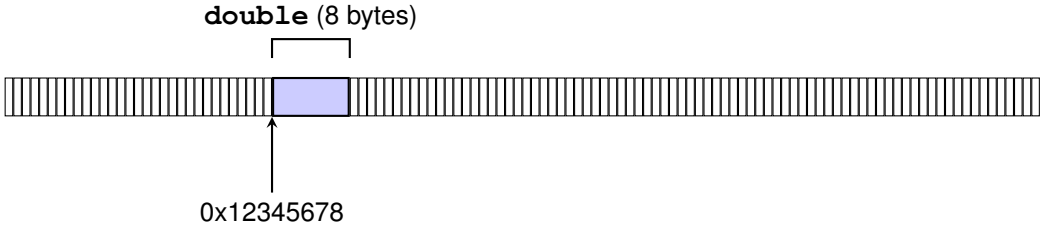
0x00000000 ..... 0xFFFFFFFF

64 bits: up to 2^{64} bytes = 16 exibytes

0x0000000000000000 ..... 0xFFFFFFFFFFFFFFFF

Memory

- in order to use chunks of memory you need to know two things:
  1. the position (**memory address**) and
  2. the size (**data type**) of a piece of memory



Memory of variables

- But haven't we used memory already?
- We didn't have to know the memory address of our variables?!?!

Memory of variables

- But haven't we used memory already?
- We didn't have to know the memory address of our variables?!?!
- **That is because the compiler takes care of variable addresses for us**
  - when it generates machine code it puts in the right memory addresses for all variables

Variable Address and Pointers

```
int i = 50;
printf("value = %d\n", i);
printf("address = %p\n", &i);
```

- Every variable has a memory address associated to it
- You can access the memory address by using the `&` (**address-of**) operator.

Variable Address and Pointers

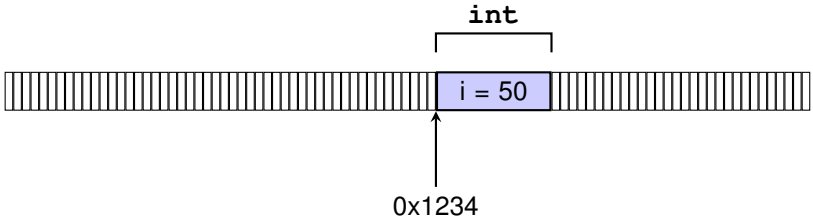
```
int i = 50;
printf("value = %d\n", i);
printf("address = %p\n", &i);
```

- Every variable has a memory address associated to it
- You can access the memory address by using the `&` (**address-of**) operator.

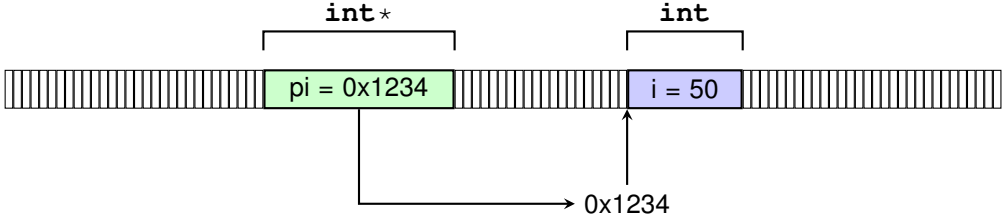
- To store a memory address with a given data type we use **pointer** data types
- A pointer data type consists of a base data type followed by a `*`

```
int* pi = &i;
printf("pointer to i = %p\n", pi);
```

Pointer to data



Pointer to data



Accessing data through a pointer

- To access the value to which the pointer points to, you need to use the dereference operator $\star$

```
int i = 50;
int* pi = &i;
printf("i   = %d\n", i);
printf("*pi = %d\n", *pi);
```

Output

```
i   = 50
*pi = 50
```

```
i = 60;
printf("i   = %d\n", i);
printf("*pi = %d\n", *pi);
```

Output

```
i   = 60
*pi = 60
```

Writing data through a pointer

- We can also change the contents of the data by assigning a new value to the dereferenced pointer variable

```
int i = 50;
int* pi = &i;

*pi = 30;

printf("i   = %d\n", i);
printf("*pi = %d\n", *pi);
```

Output

```
i   = 30
*pi = 30
```

NULL or nullptr

- pointers must store valid memory addresses
- if you want to assign an empty value use `NULL` or `nullptr` (C++11)
- this is internally the same as assigning the value 0
- by convention a `NULL` pointer signifies a pointer which points to nothing

C++98 and newer

```
int * p = NULL;
```

C++11 and newer

```
int * p = nullptr;
```

Invalid Pointers

Segmentation Fault

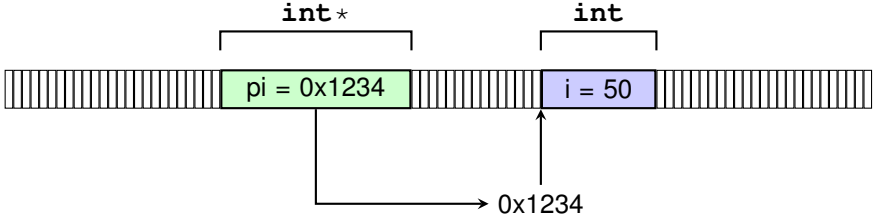
If you store a pointer to an invalid memory location and try to access that memory location, your program will crash with a *segfault* (segmentation fault). That is the best case scenario.

Pointer to Pointer

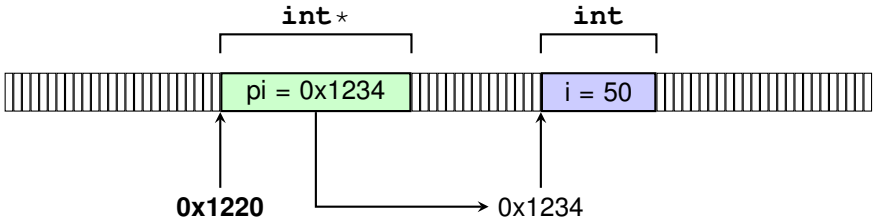
```
int i = 50;
int* pi = &i;
int** ppi = &pi;
```

- Pointers are just variables as well
- So you can have a pointer to a pointer variable
- This allows you to have as many indirections as you want

Pointer to Pointer

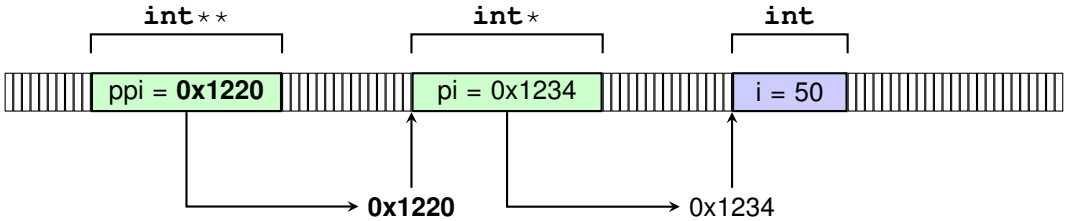


Pointer to Pointer



- a pointer is just another variable
- so it has an address itself

Pointer to Pointer



- a pointer is just another variable
- so it has an address itself
- we can save a pointer to a pointer

Dereferencing a pointer to a pointer

```
int i      = 50;
int* pi    = &i;
int** ppi = &pi;

// accessing the value i through ppi
printf("value of i: %d\n", **ppi);

// writing value of i through ppi
**ppi = 30;
printf("value of i: %d\n", i);
```

- to access the data value behind a chain of pointers you need to dereference the pointer multiple times

Output

```
value of i: 50
value of i: 30
```

How to read the pointer type:

Read the data type from right to left.

`int`$\star$ stands for:

- $\star$ = pointer of an
- **`int`** = integer

`int`$\star\star$ stands for:

- $\star$ = pointer of a
- $\star$ = pointer of an
- **`int`** = integer

Pointers

Source: https://xkcd.com/138/

Outline

Scope and Lifetime of Variables

```
int main() {
    int a = 30;
    int b = 50;

    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    return 0;
}
```

Scope and Lifetime of Variables

```
int main() {
    int a = 30;
    int b = 50;

    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    return 0;
}
```

Output

```
&a = 0x7ffd47793adc
&b = 0x7ffd47793ad8
```

- `&a > &b`
- in words: the memory address of the first variable is larger than of the second variable

Scope and Lifetime of Variables

```
int main() {
    int a = 30;
    int b = 50;
    int c = 40;

    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("&c = %p\n", &c);
    return 0;
}
```

Output

```
&a = 0x7ffd47793adc
&b = 0x7ffd47793ad8
&c = 0x7ffd47793ad4
```

- `&a > &b`
- `&b > &c`
- with each variable, their memory addresses grow towards smaller numbers

Scope and Lifetime of Variables

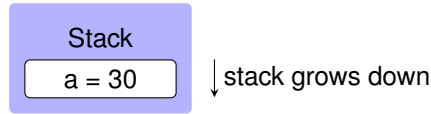
```
int main()
{
    int a = 30;
    int b = 50;

    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    return 0;
}
```

- Variables declared in a function are placed in a temporary storage location called the **stack**
- Every variable declaration pushes a new element on the stack
- when you exit a function, all of its variables are removed from the stack
- the **lifetime** of a variable spans from the declaration to the end of scope where it was defined

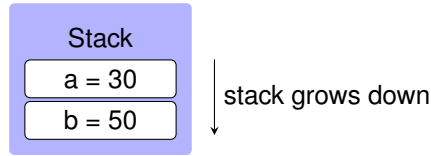
```c
int main()
{
    int a = 30;
    int b = 50;
    int c = 40;

    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("&c = %p\n", &c);
    return 0;
}
```

Stack

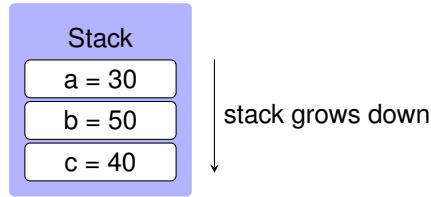
```c
int main()
{
    int a = 30;
    int b = 50;
    int c = 40;

    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("&c = %p\n", &c);
    return 0;
}
```



```c
int main()
{
    int a = 30;
    int b = 50;
    int c = 40;

    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("&c = %p\n", &c);
    return 0;
}
```



```c
int main()
{
    int a = 30;
    int b = 50;
    int c = 40;

    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("&c = %p\n", &c);
    return 0;
}
```



```c
int main()
{
    int a = 30;
    int b = 50;
    int c = 40;

    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("&c = %p\n", &c);
    return 0;
}
```

Stack

Scope

- a variable on the stack is accessible in the **scope** of a function
- at the end of that function, its scope ends
- the variable is no longer accessable and is removed

Scope

```
int main()
{
    int a = 30;
    int b = 40;

    printf("a=%d\n", a);
    printf("b=%d\n", b);

    {
        int a = 50;
        printf("a=%d\n", a);
        printf("b=%d\n", b);
    }
    printf("a=%d\n", a);
    return 0;
}
```

- functions can have multiple nested blocks
- each block is defined using curly braces and has its own scope
- { and } mark the beginning and end of a block
- a variable name can only be declared once in a scope
- but inside a nested scope you can redeclare a name; the new variable will mask the one from the outer scope

Scope

```
int main()
{
    int a = 30;
    int b = 40;

    printf("a=%d\n", a);
    printf("b=%d\n", b);

    {
        int a = 50;
        printf("a=%d\n", a);
        printf("b=%d\n", b);
    }
    printf("a=%d\n", a);
    return 0;
}
```

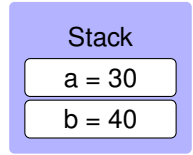
Output

```
a=30
b=40
a=50
b=40
a=30
```

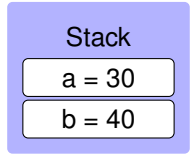
```c
int main()
{
    int a = 30;
    int b = 40;

    printf("a=%d\n", a);
    printf("b=%d\n", b);

    {
        int a = 50;
        printf("a=%d\n", a);
        printf("b=%d\n", b);
    }
    printf("a=%d\n", a);
    return 0;
}
```

Stack

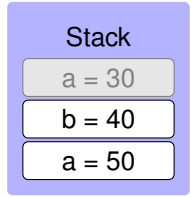
```c
int main()
{
    int a = 30;
    int b = 40;

    printf("a=%d\n", a);
    printf("b=%d\n", b);

    {
        int a = 50;
        printf("a=%d\n", a);
        printf("b=%d\n", b);
    }
    printf("a=%d\n", a);
    return 0;
}
```



```c
int main()
{
    int a = 30;
    int b = 40;

    printf("a=%d\n", a);
    printf("b=%d\n", b);

    {
        int a = 50;
        printf("a=%d\n", a);
        printf("b=%d\n", b);
    }
    printf("a=%d\n", a);
    return 0;
}
```

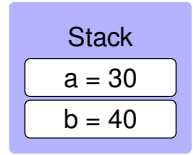


```
int main()
{
    int a = 30;
    int b = 40;

    printf("a=%d\n", a);
    printf("b=%d\n", b);

    {
        int a = 50;
        printf("a=%d\n", a);
        printf("b=%d\n", b);
    }
    printf("a=%d\n", a);
    return 0;
}
```



- the variable a declared in the inner block masks/hides the variable a from the outer block.
- in the scope of the inner block you only see two variables

```c
int main()
{
    int a = 30;
    int b = 40;

    printf("a=%d\n", a);
    printf("b=%d\n", b);

    {
        int a = 50;
        printf("a=%d\n", a);
        printf("b=%d\n", b);
    }
    printf("a=%d\n", a);
    return 0;
}
```



- after the inner block ends, the inner block's `a` variable is out of scope and is deleted from the stack.
- the original `a` variable becomes visible again

`for`-Loop Scope

```
for(int i = 0; i < 10; ++i)
{
    // loop body
    int j = 0;
    printf("%d, %d\n", i, j);
    ++j; // no effect
}
```

- variables declared in the first expression of the **`for`**-loop construct are valid for the entire loop

- variables declared inside a **`for`**-loop are only valid for one iteration

`for`-Loop Scope

```
for(int i = 0; i < 10; ++i)
{
    // loop body
    int j = 0;
    printf("%d, %d\n", i, j);
    ++j; // no effect
}
```

- variables declared in the first expression of the **`for`**-loop construct are valid for the entire loop
- An easy way to understand this is to remember that a **`for`**-loop can be built using a **`while`**-loop. These two codes are the same.

- variables declared inside a **`for`**-loop are only valid for one iteration

```
{
  int i = 0;
  while(i < 10)
  {
    // loop body
    int j = 0;
    ++j; // no effect
    ++i; // has effect
  }
}
```

`for`-Loop Scope

- You can not declare the same variable in the same scope more than once
- But because a **`for`**-Loop is a block and therefore has its own scope you can write this:

```
int i = 50;

for(int i = 0; i < 5; ++i) {
    printf("i = %d (first loop)\n", i);
}

for(int i = 0; i < 10; ++i) {
    printf("i = %d (second loop)\n", i);
}

printf("final value of i = %d\n", i);
```

`for`-Loop Scope

Output

```
i = 0 (first loop)
i = 1 (first loop)
i = 2 (first loop)
i = 3 (first loop)
i = 4 (first loop)
i = 0 (second loop)
i = 1 (second loop)
i = 2 (second loop)
i = 3 (second loop)
i = 4 (second loop)
i = 5 (second loop)
i = 6 (second loop)
i = 7 (second loop)
i = 8 (second loop)
i = 9 (second loop)
final value of i = 50
```

Stack frames

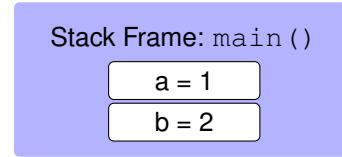
- so far we only looked at variables defined in one function
- we said that those variables are placed on the stack
- now that we are going to use more than one function, let's be more precise with our naming
- the part of the stack a function *sees* is called a **stack frame**

Calling a function

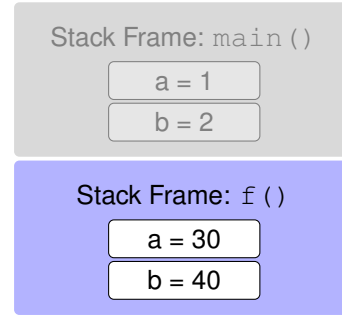
```
void g() {
}

void f() {
    int a = 30;
    int b = 40;
    printf("inside: a=%d, b=%d\n", a, b);
    g();
}

int main() {
    int a = 1;
    int b = 2;

    printf("before: a=%d, b=%d\n", a, b);
    f();
    printf("after: a=%d, b=%d\n", a, b);
    return 0;
}
```

- when you call a function you create a new empty stack frame
- that stack frame is the part of the stack this function is allowed to see
- each function only sees variables in its scope and puts them in its current stack frame
- variables from one stack frame are not visible in another

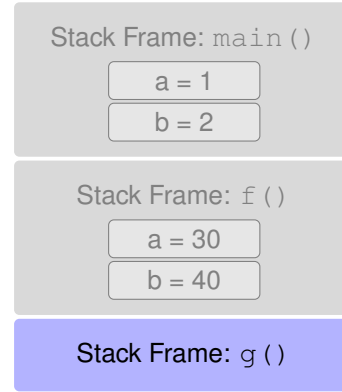
```c
void g() {
}

void f() {
    int a = 30;
    int b = 40;
    printf("inside: a=%d, b=%d\n", a, b);
    g();
}

int main() {
    int a = 1;
    int b = 2;

    printf("before: a=%d, b=%d\n", a, b);
    f();
    printf("after: a=%d, b=%d\n", a, b);
    return 0;
}
```



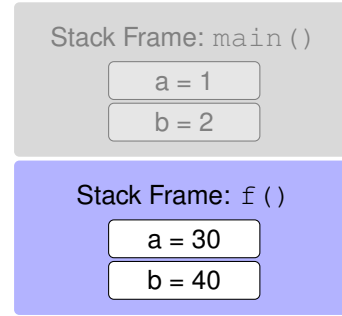
```c
void g() {
}

void f() {
    int a = 30;
    int b = 40;
    printf("inside: a=%d, b=%d\n", a, b);
    g();
}

int main() {
    int a = 1;
    int b = 2;

    printf("before: a=%d, b=%d\n", a, b);
    f();
    printf("after: a=%d, b=%d\n", a, b);
    return 0;
}
```



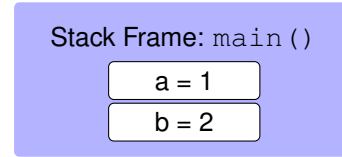
```c
void g() {
}

void f() {
    int a = 30;
    int b = 40;
    printf("inside: a=%d, b=%d\n", a, b);
    g();
}

int main() {
    int a = 1;
    int b = 2;

    printf("before: a=%d, b=%d\n", a, b);
    f();
    printf("after: a=%d, b=%d\n", a, b);
    return 0;
}
```



```c
void g() {
}

void f() {
    int a = 30;
    int b = 40;
    printf("inside: a=%d, b=%d\n", a, b);
    g();
}

int main() {
    int a = 1;
    int b = 2;

    printf("before: a=%d, b=%d\n", a, b);
    f();
    printf("after: a=%d, b=%d\n", a, b);
    return 0;
}
```



```c
void g() {
}

void f() {
    int a = 30;
    int b = 40;
    printf("inside: a=%d, b=%d\n", a, b);
    g();
}

int main() {
    int a = 1;
    int b = 2;

    printf("before: a=%d, b=%d\n", a, b);
    f();
    printf("after: a=%d, b=%d\n", a, b);
    return 0;
}
```



Calling a function with parameters

```
void f(int p1, int p2) {
    int a = 30;
    printf("inside: p1=%d, p2=%d\n", p1, p2);
    printf("inside: a=%d\n", a);
}

int main() {
    int a = 1;

    f(42, a);

    return 0;
}
```

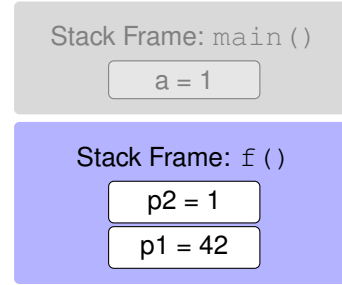
- if you call a function with parameters, those parameters are put on the empty new stack frame
- parameter values are **copied** into the new stack frame
- this is called *call-by-value* or *pass-by-value*

```c
void f(int p1, int p2) {
    int a = 30;
    printf("inside: p1=%d, p2=%d\n", p1, p2);
    printf("inside: a=%d\n", a);
}

int main() {
    int a = 1;

    f(42, a);

    return 0;
}
```

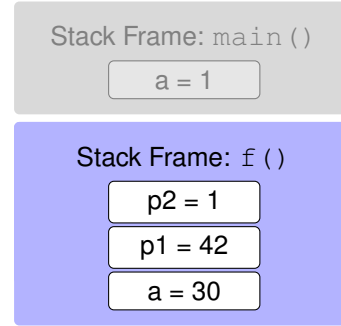
Stack Frame: `main()`

a = 1

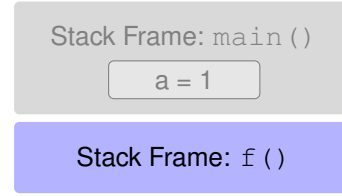
```c
void f(int p1, int p2) {
    int a = 30;
    printf("inside: p1=%d, p2=%d\n", p1, p2);
    printf("inside: a=%d\n", a);
}

int main() {
    int a = 1;

    f(42, a);

    return 0;
}
```



```c
void f(int p1, int p2) {
    int a = 30;
    printf("inside: p1=%d, p2=%d\n", p1, p2);
    printf("inside: a=%d\n", a);
}

int main() {
    int a = 1;

    f(42, a);

    return 0;
}
```



```c
void f(int p1, int p2) {
    int a = 30;
    printf("inside: p1=%d, p2=%d\n", p1, p2);
    printf("inside: a=%d\n", a);
}

int main() {
    int a = 1;

    f(42, a);

    return 0;
}
```



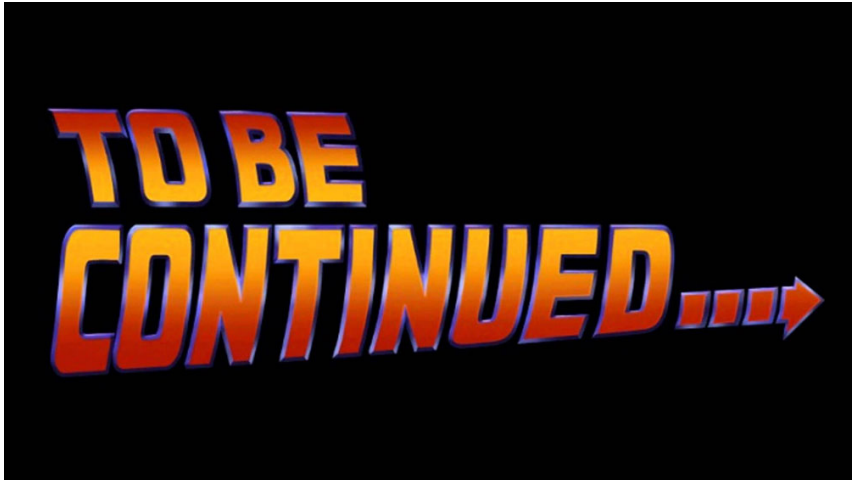
```c
void f(int p1, int p2) {
    int a = 30;
    printf("inside: p1=%d, p2=%d\n", p1, p2);
    printf("inside: a=%d\n", a);
}

int main() {
    int a = 1;

    f(42, a);

    return 0;
}
```

Stack Frame: `main()`

a = 1

- Q: What kind of parameter can we pass to a function to access variable from outside its own stack frame?

- Q: What kind of parameter can we pass to a function to access variable from outside its own stack frame?
- A: a pointer to that variable


TO BE
CONTINUED...