Signals, Instruments, and Systems – W6

C Programming Refresher

Advanced topics

Outline

- Compiled vs interpreted languages
- Compilation
- Pointers
- Memory management
- Debugging with gdb

Compiled vs interpreted languages

Compiled Language

Code is directly translated (through a **compiler**) into binary that can be executed by the machine.

- ✓ Direct access machine resources (memory, processes)
- ✓ More efficient
- ✓ Faster

Examples: C, C++

Interpreted Language

Need for a running engine (e.g., JVM for Java codes) to be translated to binary (**usually at run-time**).

- ✓ Usually platform independent
- ✓ Easier to code
- ✓ High-level control of resources

Examples: Java, Python, Matlab

Main differences between C and Matlab

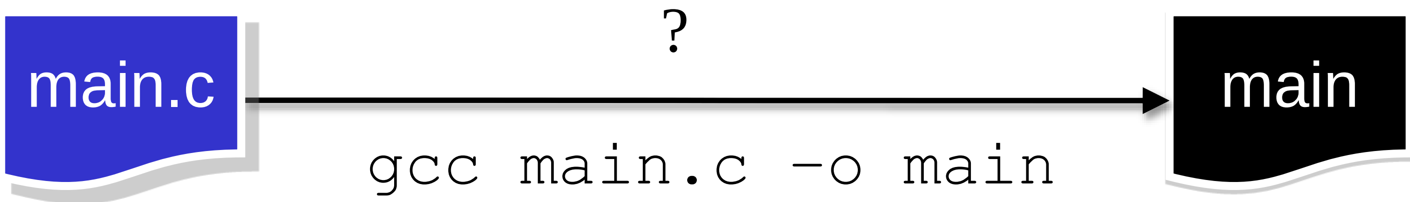
- Matlab is an interpreted language
- Matlab is optimized for matrix operations
- Syntax differences (loops, functions, etc…)
- No variable declarations

Object Oriented Languages

- Object Oriented programming = data is grouped into “objects” that can have properties and/or methods (functions)

- C is NOT an object-oriented language, C++ is

Compilation



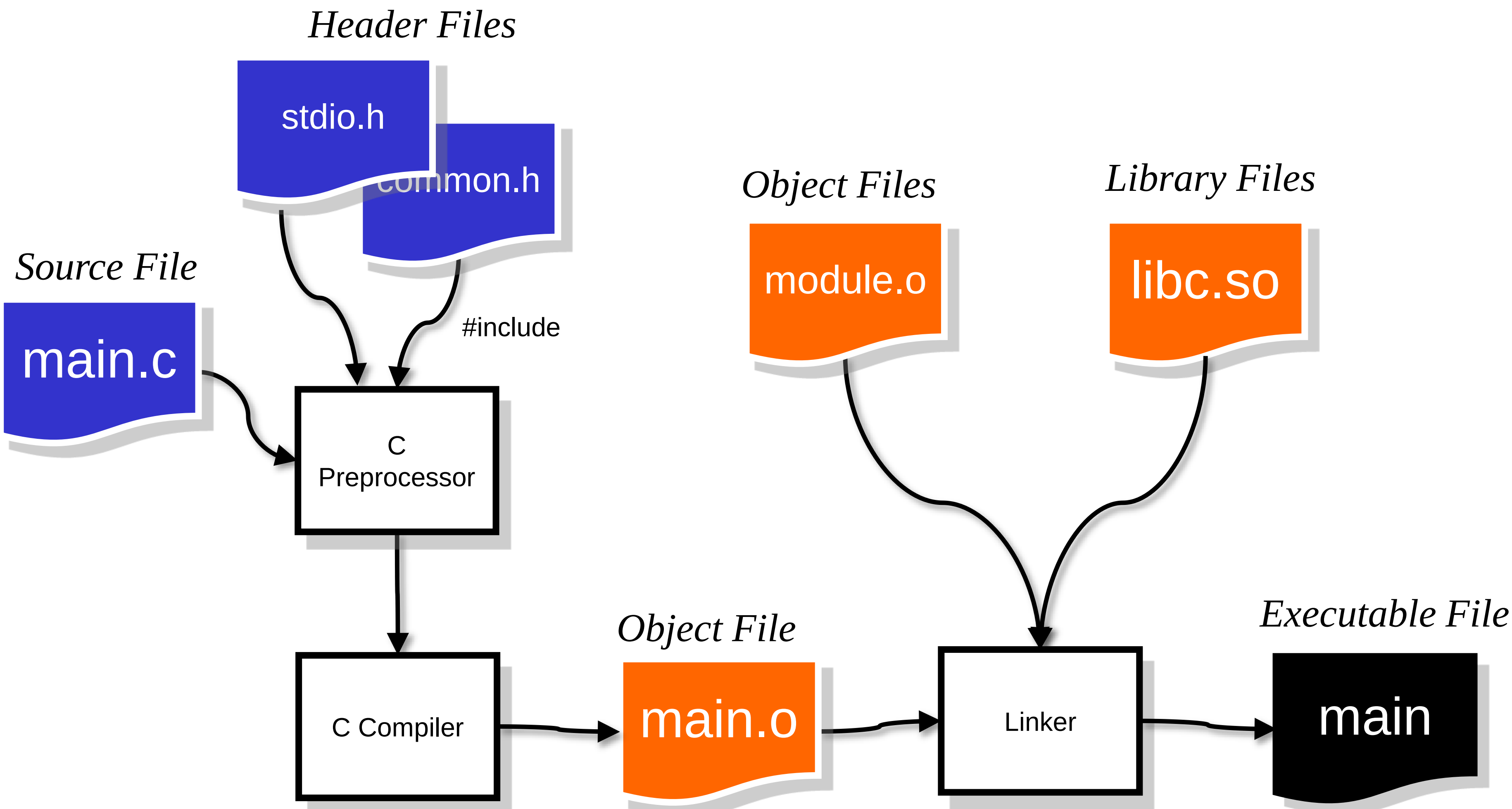
source file

executable (binary)

```
int main() {
  int a = 5;
  double b = 4.3;
  return a * b;
}
```

```
10100101010010
10100101001010
10010100101001
00010101001111
00100101010100
```

The C Compiler Pipeline



“.h” vs. “.c”

- Usually header files (“.h” files) should contain all the necessary functions, structures, typedef and enum **declarations** such that **another** programmer can use your code without having to look at your c file.
- C files contain the actual implementation and “hidden” declarations.

Libraries

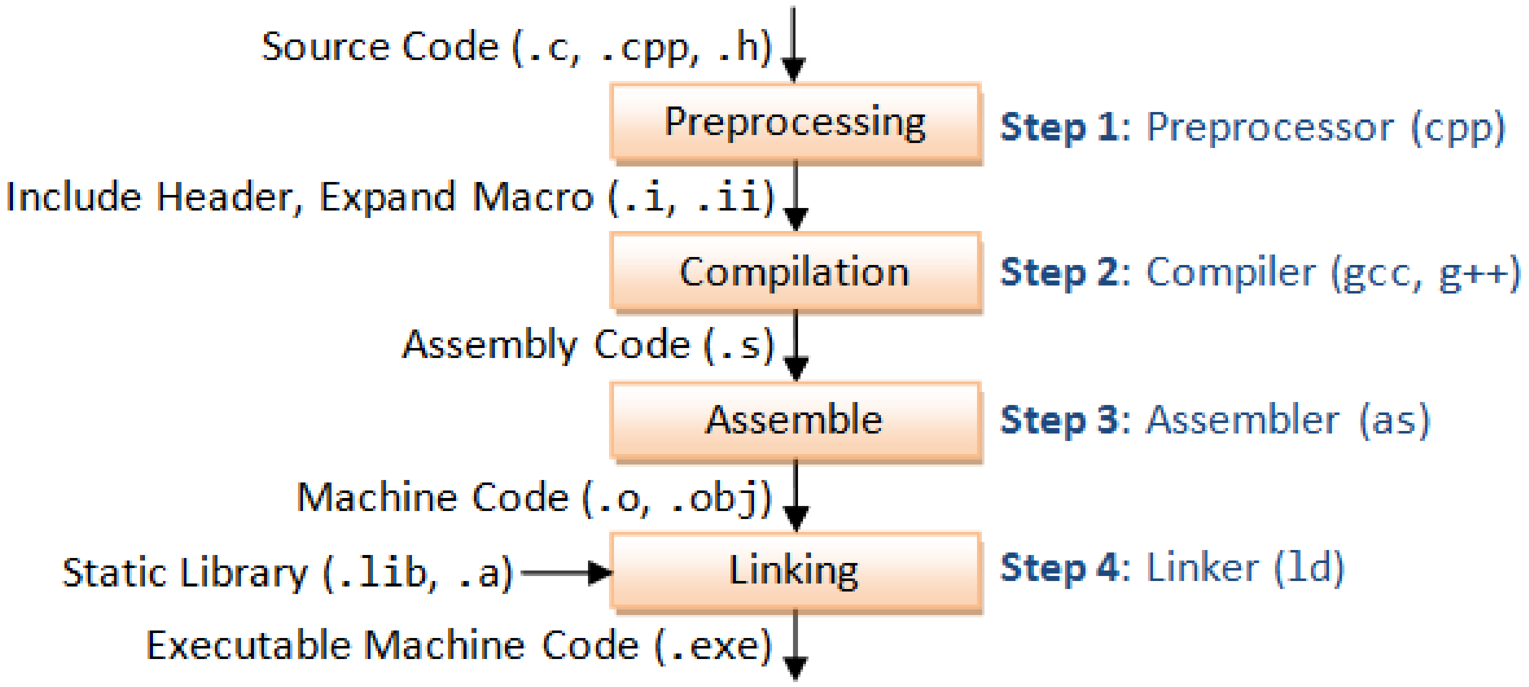
- Libraries provide special functionality in the form of collections of ready-made functions:

Library:	**Example:**
`stdio.h`	`printf(const char* format,…)`
`math.h`	`sqrt(double x)`
`time.h`	`gettimeofday()`
`stdlib.h`	`rand()`

Usage:

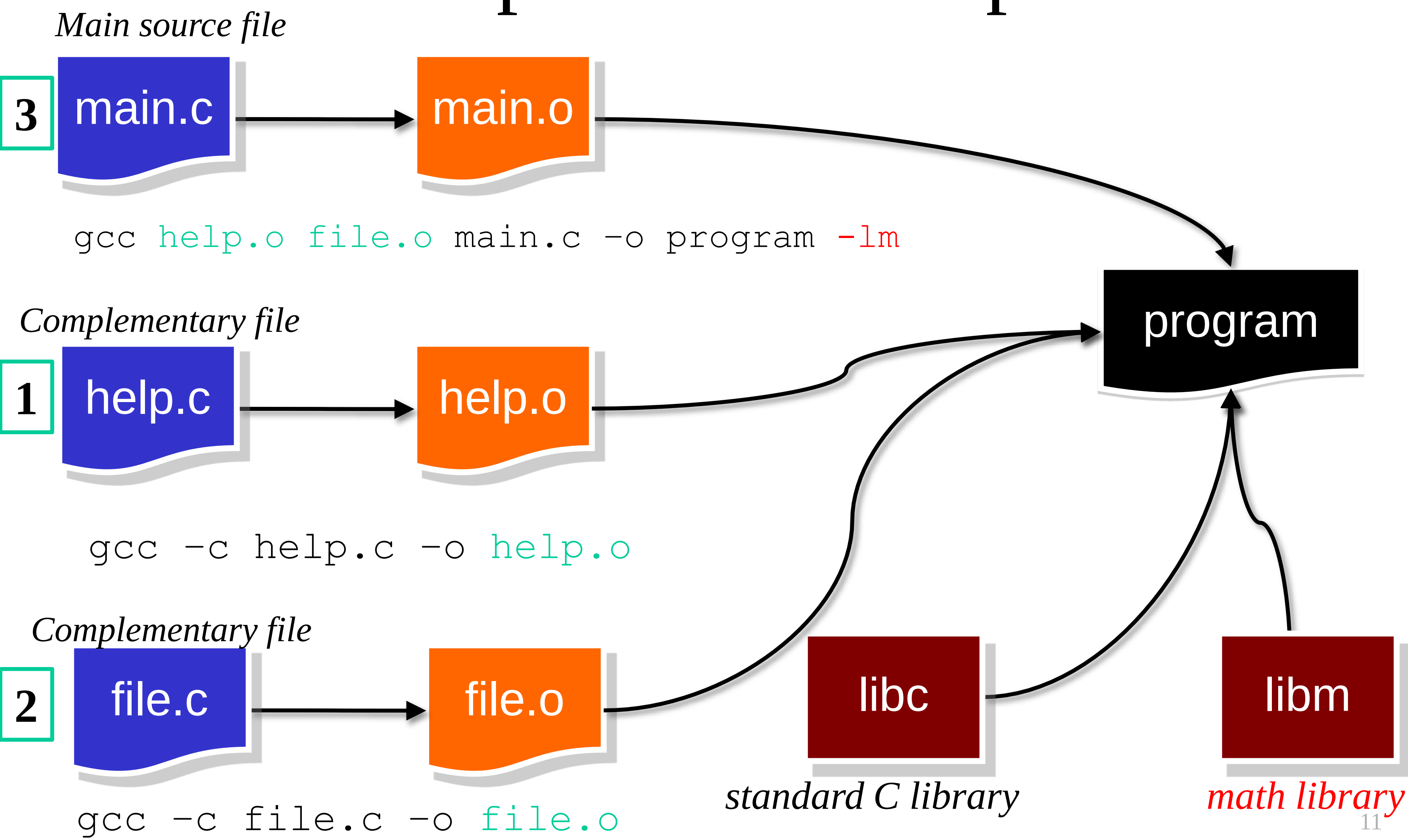
```
#include <stdlib.h>
#include "my_library.h"
```
: your own collection of function declarations

Compilation process



https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

Compilation Example



Makefile: Example

```
CC = gcc
LDLIBS = -lm
all: program

program: main.o help.o file.o



clean:
    rm -rf *.o program
```

- compiler
- additional library
- targets
- label
- [TAB] !!
  [TAB] `rm -rf *.o main`

Note: Run `make clean all` for a totally new compilation

Warning

- Build commands explained here are for Unix-like systems (i.e. Ubuntu, MacOS* etc.)
- For Windows, different commands are utilized
- All will be explained in labs/guidelines

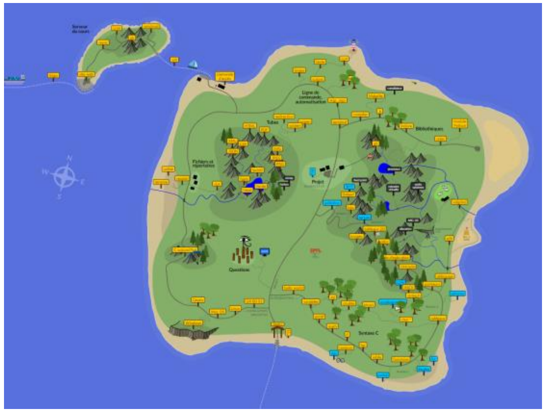
Tool to be used

	Ubuntu	MacOS	Windows
Compiler	*gcc*	*gcc (Clang)*	*gcc (Mingw-w64)*
Debugger	*gdb*	*lldb*	*gdb (Mingw-w64)*
IDE	*Geany, Webots*	*Geany, Webots*	*Geany, Webots*

You need to review:

- Variables and constants (types, sign, global, local etc.)
- Operators (arithmetic, unary, bitwise etc.)
- Controlling execution flow (if, switch, while, do-while, for etc.)
- Standard libraries (stdin, stdout etc.)
- Functions
- Arrays, structure, strings, type definitions
- Preprocessor commands

https://coursc.ch/



Argument passing in C

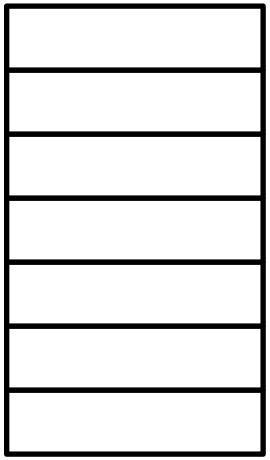
- Reminder:
  Arguments are always passed *by value* in C function calls! This means that **local copies** of the values of the arguments are passed to the routines!

```
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a, b);

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

  return 0;
}
```

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 5, b = 7
```

What happens?

```
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

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

  return 0;
}
```



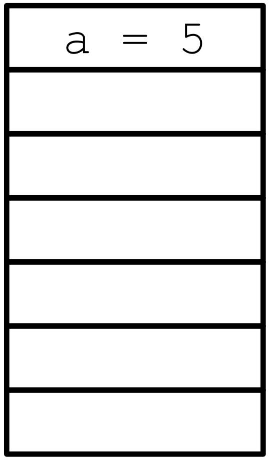
Computer memory

Output:

```
computer:~> ./exchange
```

What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

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

  return 0;
}
```



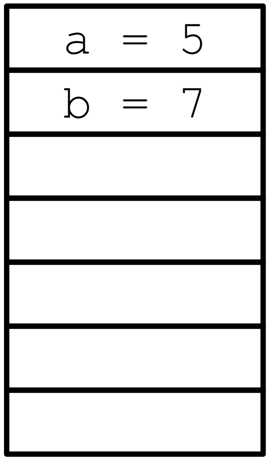
Computer memory

Output:

```
computer:~> ./exchange
```

What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

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

  return 0;
}
```



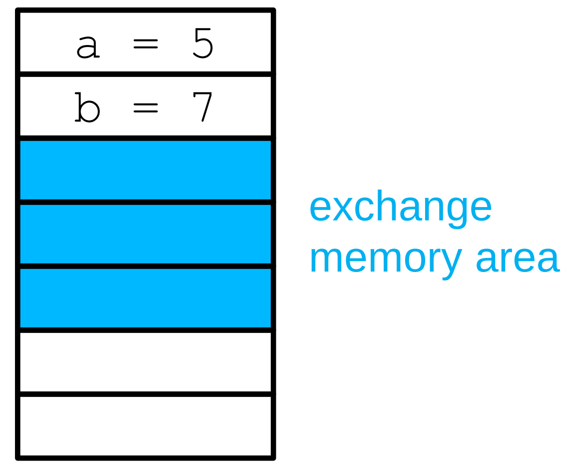
Computer memory

Output:

```
computer:~> ./exchange
```

What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

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

  return 0;
}
```



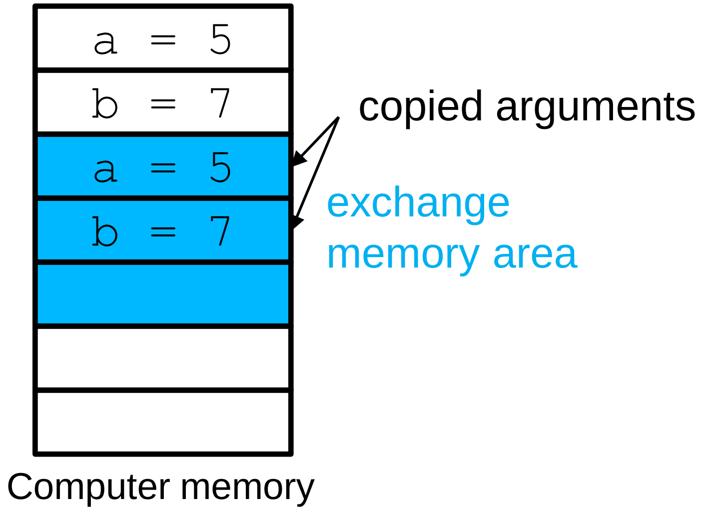
Computer memory

Output:

```
computer:~> ./exchange
```

What happens?

```
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

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

  return 0;
}
```

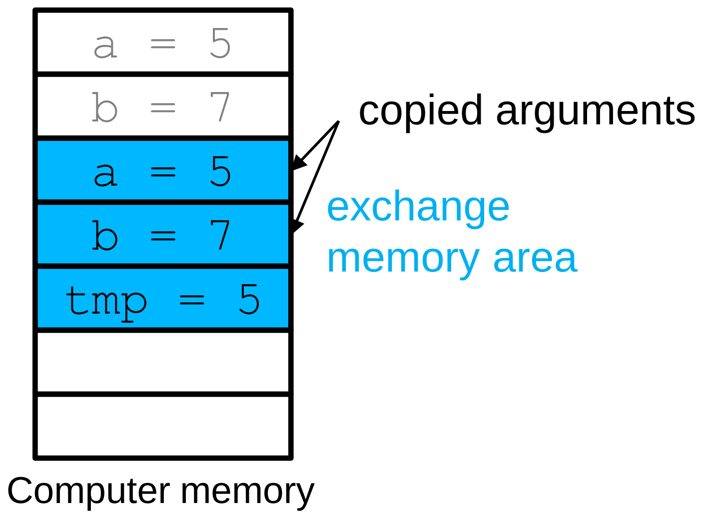


Output:

```
computer:~> ./exchange
```

What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

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

  return 0;
}
```

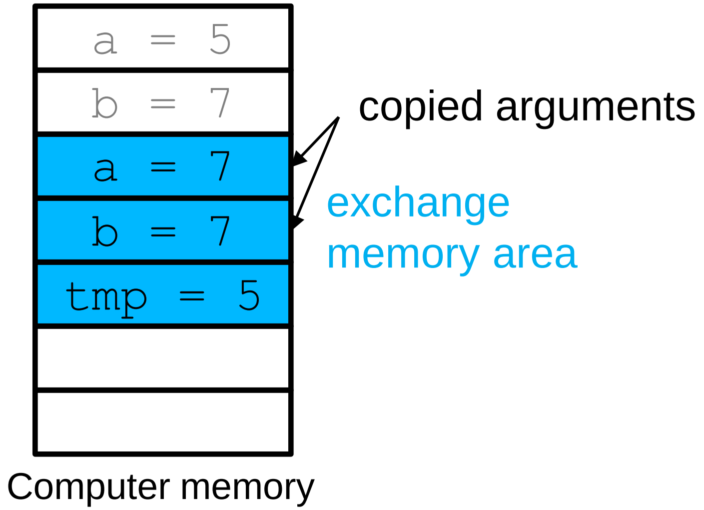


Output:

```
computer:~> ./exchange
```

What happens?

```
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

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

  return 0;
}
```

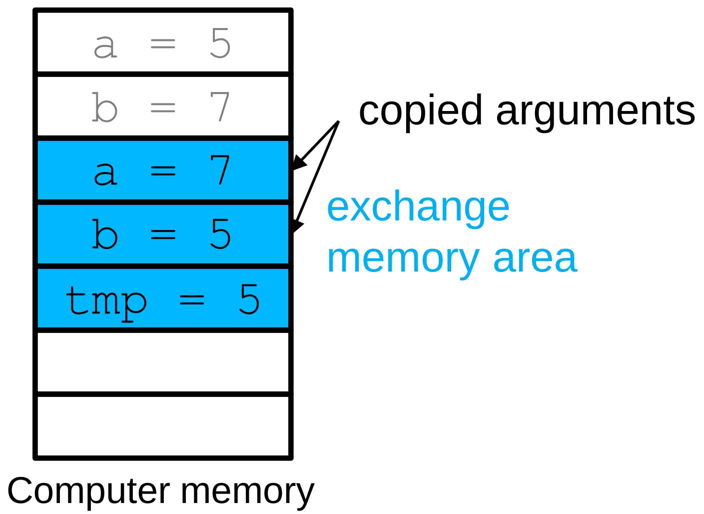


Output:

```
computer:~> ./exchange
```

What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

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

  return 0;
}
```

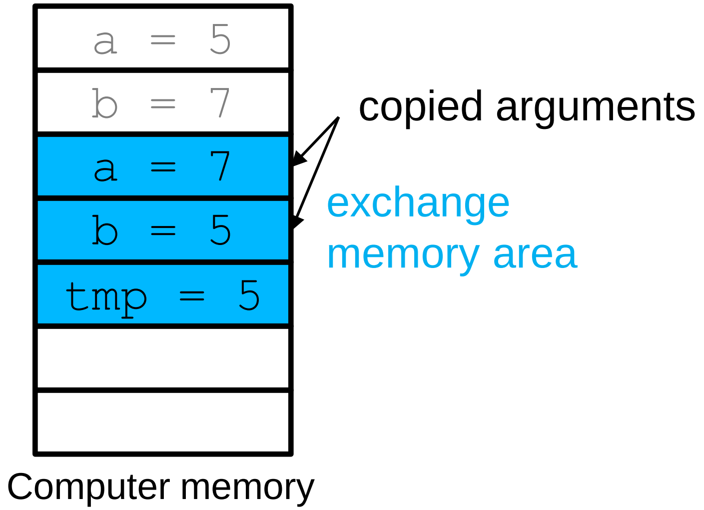


Output:

```
computer:~> ./exchange
```

What happens?

```
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

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

  return 0;
}
```

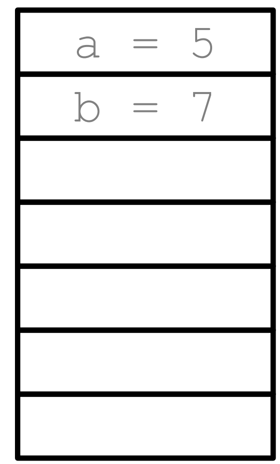


Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
```

What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

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

  return 0;
}
```



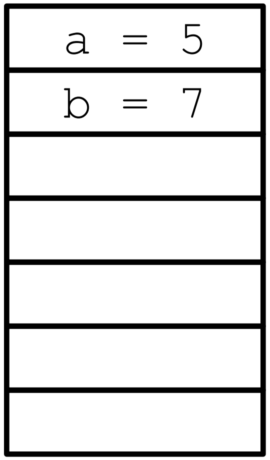
Computer memory

Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
```

What happens?

```c
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

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

  return 0;
}
```



Computer memory

Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 5, b = 7
```

Pointers

```
int i;

int* pi;
```

Pointers

```
int i;

int* pi;
```

=

```
int *pi;
```

Pointers

```
float f;

float* pf;
=
float *pf;
```

Pointers

Address	Content
5460	
5464	
5468	-

Pointers

```
int a = 5;
```

Address	Content
5460	
5464	
5468	

Pointers

```
int a = 5;
```

Address	Content
5460	a = 5
5464	
5468	

Pointers

```
int a = 5;

int b = 7;
```

Address	Content
5460	a = 5
5464	
5468	

Pointers

```
int a = 5;

int b = 7;
```

Address	Content
5460	a = 5
5464	b = 7
5468	

Pointers

```
int a = 5;

int b = 7;

int* pa;
```

Address	Content
5460	a = 5
5464	b = 7
5468	

Pointers

```
int a = 5;

int b = 7;

int* pa = &a;
```

Address	Content
5460	a = 5
5464	b = 7
5468	-

Pointers

```
int a = 5;

int b = 7;

int* pa = &a;
```

address-of operator

Address	Content
5460	a = 5
5464	b = 7
5468	

Pointers

```
int a = 5;

int b = 7;

int* pa = &a;
```

address-of operator

Address	Content
5460	a = 5
5464	b = 7
5468	pa = 5460

Pointers

Address	Content
5460	`a = 5`
5464	`b = 7`
5468	`pa = 5460`

Pointers

```
pa = &b;
```

Address	Content
5460	`a = 5`
5464	`b = 7`
5468	`pa = 5460`

Pointers

```
pa = &b;
```

Address	Content
5460	a = 5
5464	b = 7
5468	pa = 5464

Pointers

```
pa = &b;

*pa = 42;
```

Address	Content
5460	a = 5
5464	b = 7
5468	pa = 5464

Pointers

```
pa = &b;

*pa = 42;
```

↑ indirection operator

Address	Content
5460	a = 5
5464	b = 7
5468	pa = 5464

Pointers

```
pa = &b;

*pa = 42;
```

↑ indirection operator

Address	Content
5460	a = 5
5464	b = 42
5468	pa = 5464

Pointers

```
pa = &b;

*pa = 42;

a = *pa;
```

Address	Content
5460	a = 5
5464	b = 42
5468	pa = 5464

Pointers

```
pa = &b;

*pa = 42;

a = *pa;
```

Address	Content
5460	a = 42
5464	b = 42
5468	pa = 5464

Anti-confusion tips

- Which one to choose ? `&var`, `var`, or `*var` ?

1. ‘`&var`’ can never come before an ‘`=`‘
2. Look for the type:

```
float i;
float *p;
p = &i;
*p = 2.4;
```

Anti-confusion tips

- Which one to choose ? `&var`, `var`, or `*var` ?

1. ‘`&var`’ can never come before an ‘`=`‘
2. Look for the type:

```
float i;
float *p;
p = &i;
*p = 2.4;
```

– p is a pointer to float, *p is a float.

Anti-confusion tips

- Which one to choose ? `&var`, `var`, or `*var` ?

1. ‘`&var`’ can never come before an ‘`=`‘
2. Look for the type:

```
float i;
float *p;
p = &i;
*p = 2.4;
```

   – p is a pointer to float, *p is a float.

Anti-confusion tips

- Which one to choose ? `&var`, `var`, or `*var` ?

1. ‘`&var`’ can never come before an ‘`=`‘
2. Look for the type:

```
float i;
float *p;
p = &i;
*p = 2.4;
```

   – p is a pointer to float, *p is a float.

Argument passing in C

- Arguments are always passed *by value* in C function calls! This means that **local copies** of the values of the arguments are passed to the routines!

```
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf("Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a, b);

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

  return 0;
}
```

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 5, b = 7
```

How to solve the problem?

- By using **pointers**, i.e. variables that contain the address of another variable!

```
#include <stdio.h>

void exchange(int* pa, int* pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a, &b);

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

  return 0;
}
```

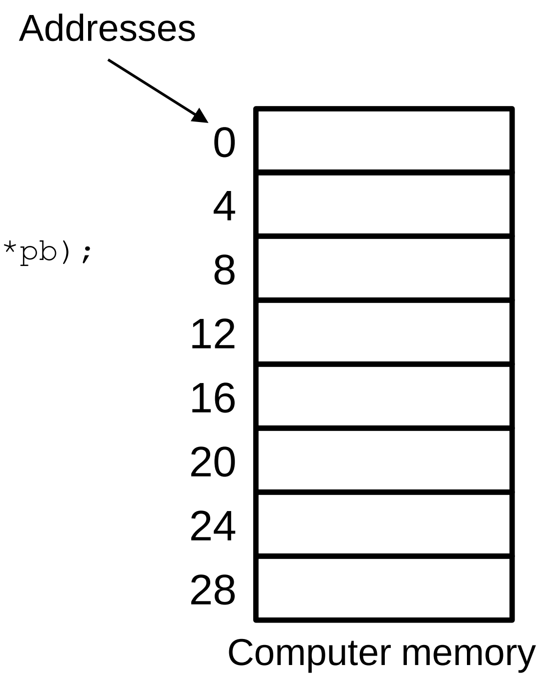
Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 7, b = 5
```

`int* pa` and **`int* pb`** are pointers!

What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

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

  return 0;
}
```

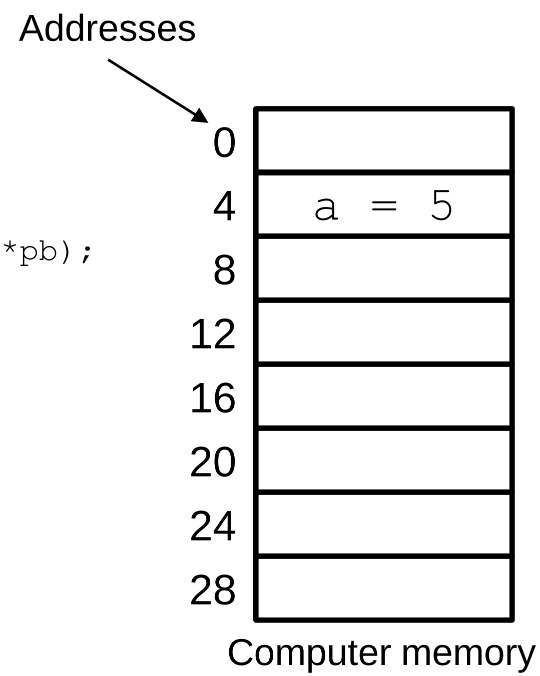


Output:

```
computer:~> ./exchange
```

What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

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

  return 0;
}
```

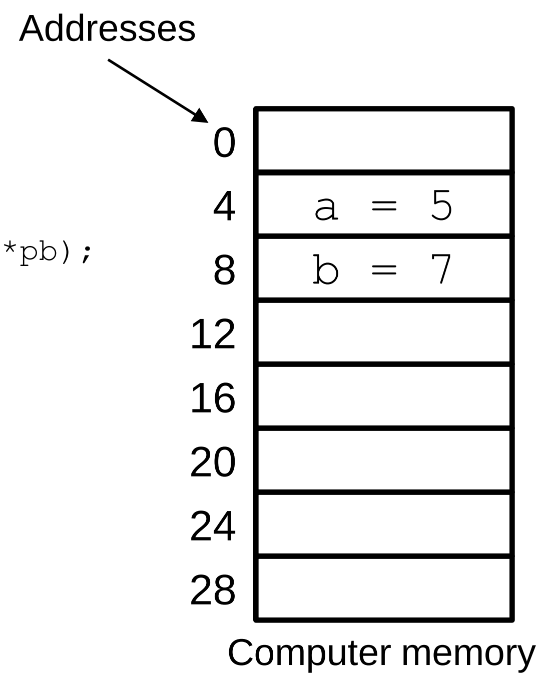


Output:

```
computer:~> ./exchange
```

What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

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

  return 0;
}
```

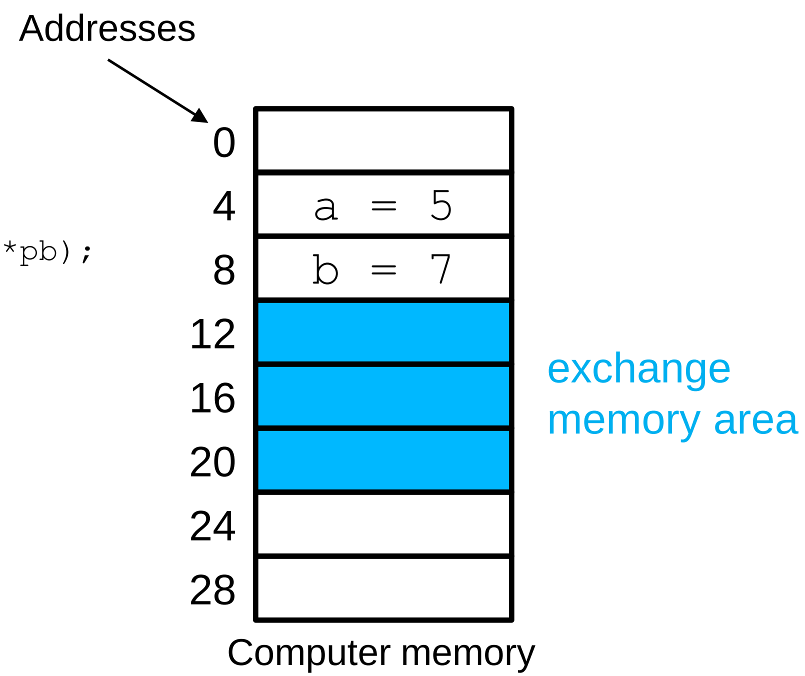


Output:

```
computer:~> ./exchange
```

What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

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

  return 0;
}
```

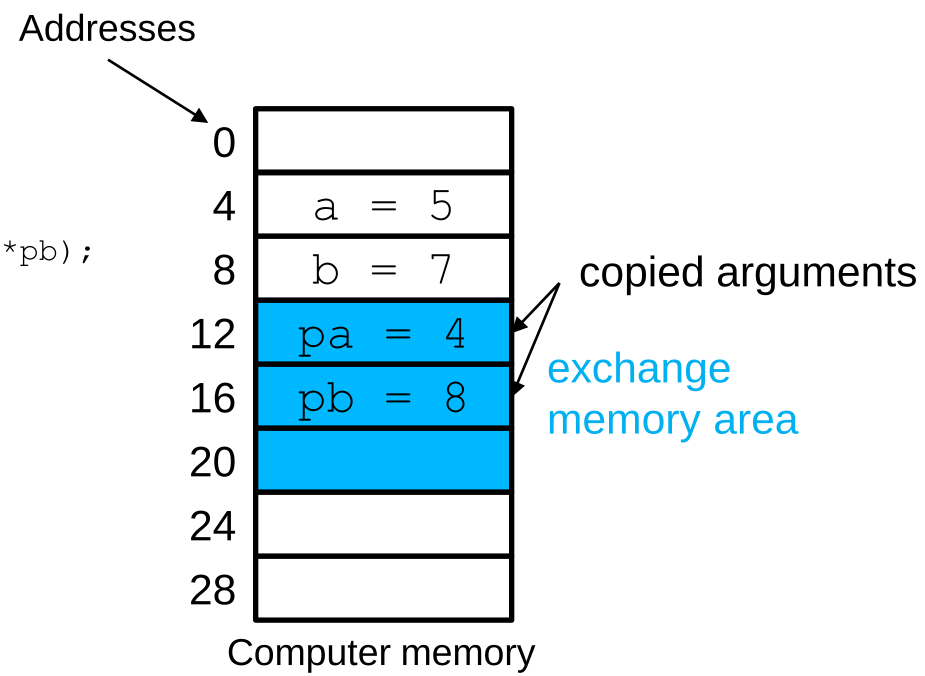


Output:

```
computer:~> ./exchange
```

What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

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

  return 0;
}
```

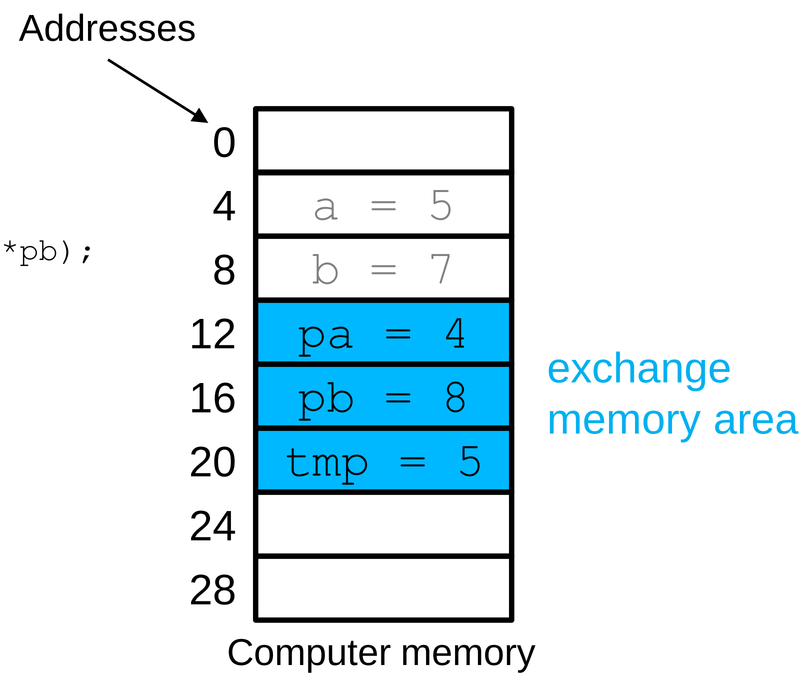


Output:

```
computer:~> ./exchange
```

What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

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

  return 0;
}
```

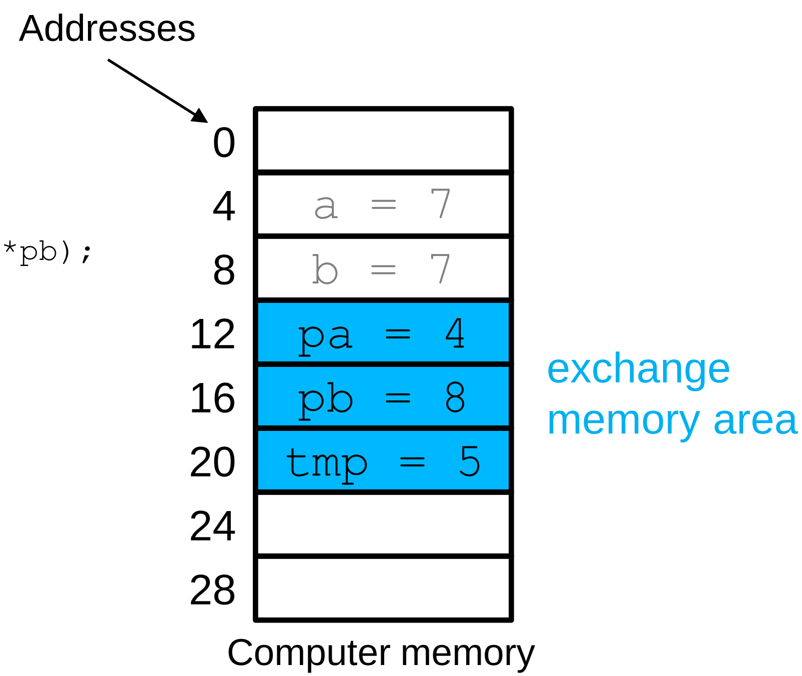


Output:

```
computer:~> ./exchange
```

What happens now?

```c
#include <stdio.h>

void exchange(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

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

  return 0;
}
```

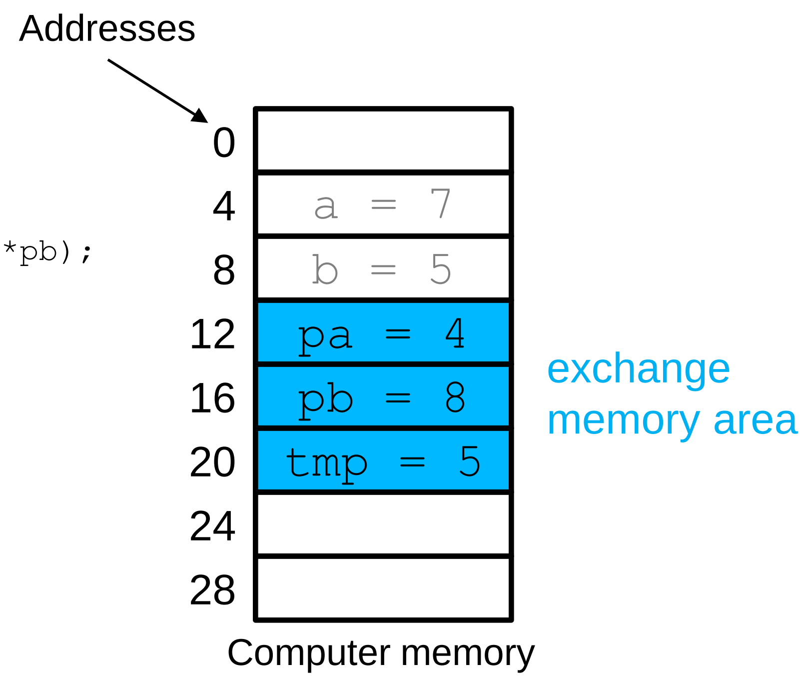


Output:

```
computer:~> ./exchange
```

What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

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

  return 0;
}
```

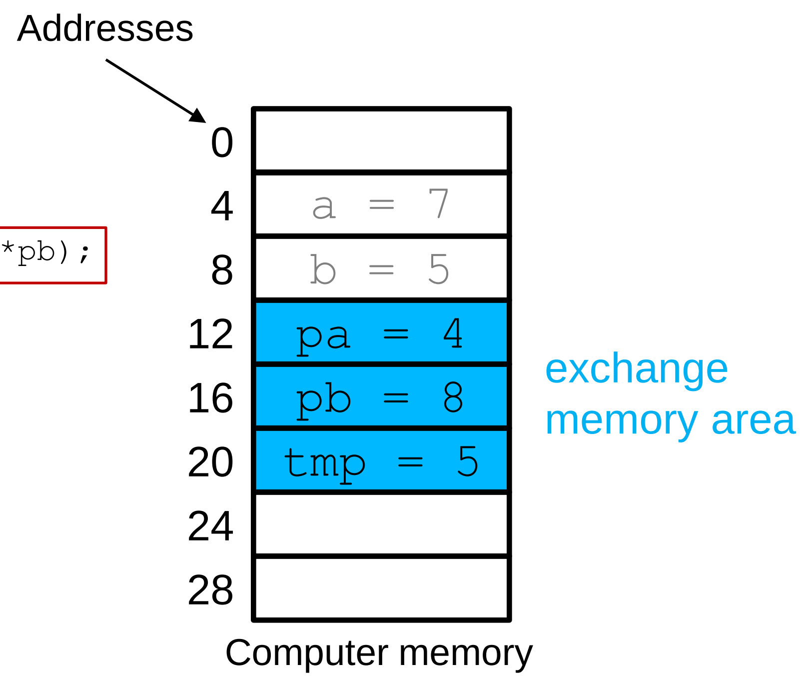


Output:

```
computer:~> ./exchange
```

What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

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

  return 0;
}
```

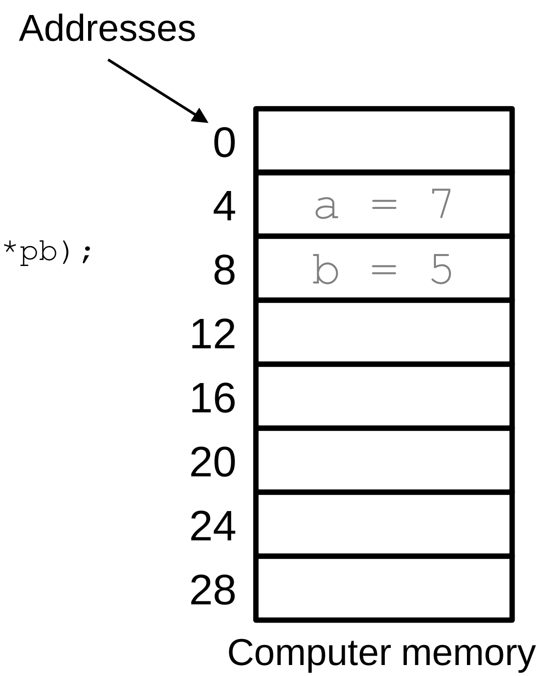


Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
```

What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

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

  return 0;
}
```

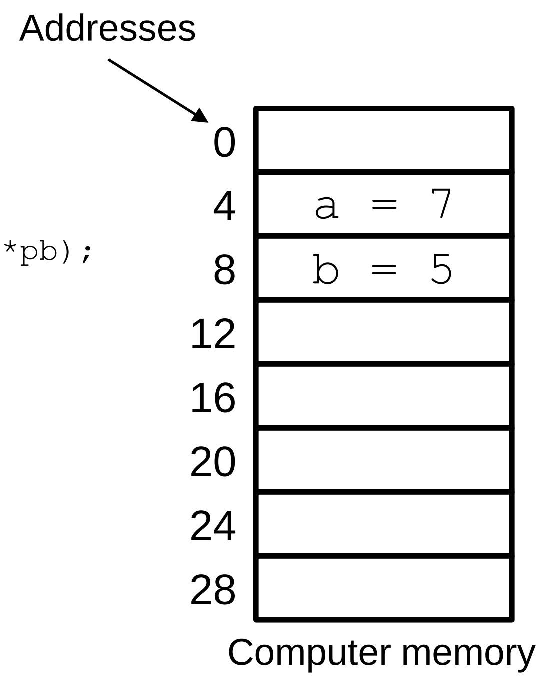


Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
```

What happens now?

```
#include <stdio.h>

void exchange(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("Exchange: a = %d, b = %d\n", *pa, *pb);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

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

  return 0;
}
```



Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 7, b = 5
```

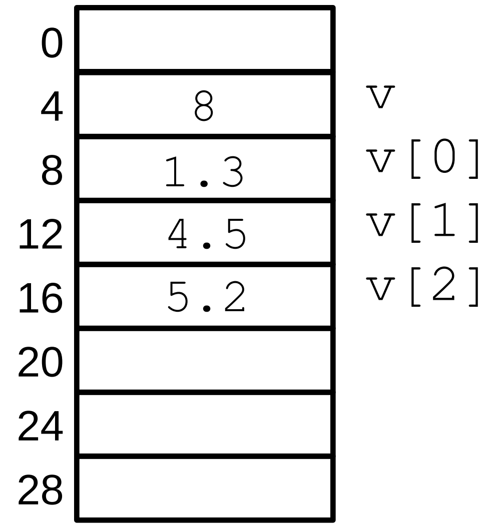
Arrays

- Arrays and pointers are closely related.

```
float v[3];
v[0] = 1.3;
v[1] = 4.5;
v[2] = 5.2;
```

- `v` points to the first element of the array

	Type	Address	Value
v	float*	4	8
v[1]	float	v+1=12	4.5



Computer memory

- `v` is the same as `&(v[0])`
- `v[0]` is the same as `*(v+0)`
- `v[1]` is the same as `*(v+1)`

Passing an array to a function

```
#include <stdio.h>

#define SIZE 3

void g(int* array_p, int const size) {
    int i;

    for (i = 0; i < size; ++i) {
        array_p[i] = 2 * (i+1);
    }
}

int main(void) {
    int i;
    int array[SIZE] = {0, 0, 0} ;

    g(array, SIZE);

    for (i = 0; i < SIZE; ++i) {
        printf("%d:%d ", i, array[i]);
    }

    return 0;
}
```

- The two variables **array_p** and **array** are not the same (**array_p** is a pointer to the first element of **array**)!
- For the purpose of modifying the array from the function g(), **array_p** acts the same as **array**
- Here is the output of the program:

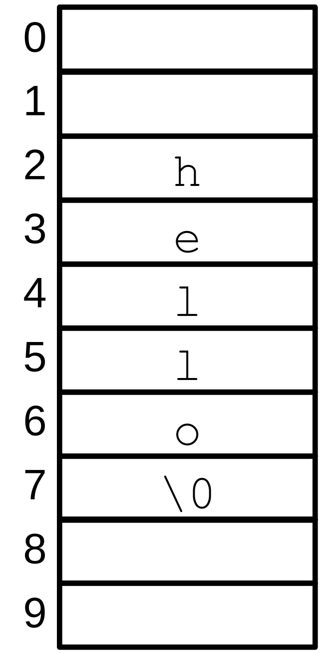
```
computer:~> gcc -o array2fun array2fun.c
computer:~> ./array2funcomputer:~> 0:2
1:4 2:6
```

Strings

- There is no string type in C. Instead, we use arrays of `char`, i.e. the type `char*`.

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

	Type	Address	Value
str	char*		2
str[4]	char	str+4	'o'
str[2]	char	str+2	'l'



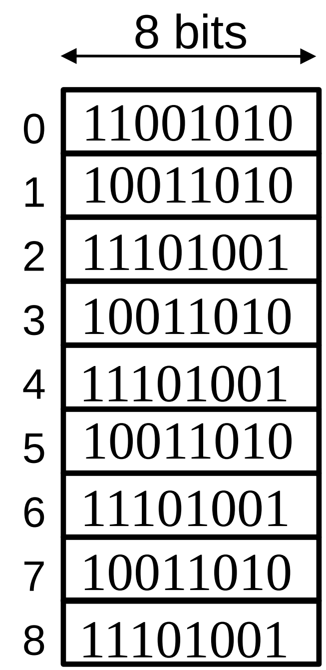
Computer memory

- You can use the `printf` to print out chains of characters. It will read up to the character `'\0'`.

```
printf("%s",str);     →  computer:~> hello
printf("%s",str+3);   →  computer:~> lo
```

Memory: a more realistic approach

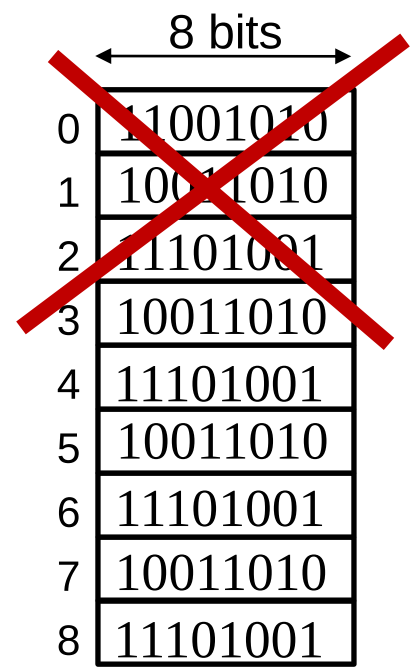
- In a real computer, memory is organized into blocks of 8 bits, called **bytes**.
- On most modern computers, each byte has its own address.
- Memory is **limited,** not only in terms of the number of RAM modules that are installed, but also in terms of the number of addresses available.
- Furthermore, a program is not allowed to use (read and/or write) all bytes: some are reserved by the operating system. If you try to access them (using a pointer), your program will crash (segmentation fault or bus error).



8-bit computer memory

Memory: a more realistic approach

- In a real computer, memory is organized into blocks of 8 bits, called **bytes**.
- On most modern computers, each byte has its own address.
- Memory is **limited,** not only in terms of the number of RAM modules that are installed, but also in terms of the number of addresses available.
- Furthermore, a program is not allowed to use (read and/or write) all bytes: some are reserved by the operating system. If you try to access them (using a pointer), your program will crash (segmentation fault or bus error).



8-bit computer memory

```
int *p = 1;
*p = 0;
```

→ segmentation fault (trying to write at address 1)

The size of the data types

- Each data type requires a certain number of bytes to be stored in memory, and this size can change as a function of the operating system (Windows, Linux, etc.) and the architecture of the system.
- The function `sizeof(`*`type`*`)` returns the size of the data type (in bytes).

```
printf("%d",sizeof(char));     /*  prints 1 */
printf("%d",sizeof(short));    /*  prints 2 */
printf("%d",sizeof(int));      /*  prints 4 */
printf("%d",sizeof(long));     /*  prints 4 */
printf("%d",sizeof(float));    /*  prints 4 */
printf("%d",sizeof(double));   /*  prints 8 */
```

The size of pointers

- **Reminder**: a pointer is a **variable** that contains **the address of another variable**.
- Therefore, the size of any pointer is **constant**, regardless of the data type that it points to (since it contains only the address of the variable, which does not depend on its type, obviously).

```
printf("%d",sizeof(char*));     /*  prints 4 */
printf("%d",sizeof(short*));    /*  prints 4 */
printf("%d",sizeof(int*));      /*  prints 4 */
printf("%d",sizeof(long*));     /*  prints 4 */
printf("%d",sizeof(float*));    /*  prints 4 */
printf("%d",sizeof(double*));   /*  prints 4 */
```

On a 32-bit computer

The size of pointers

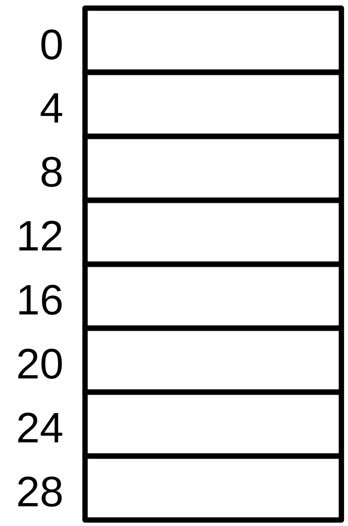
- **Reminder**: a pointer is a **variable** that contains **the address of another variable**.
- Therefore, the size of any pointer is **constant**, regardless of the data type that it points to (since it contains only the address of the variable, which does not depend on its type, obviously).

```
printf("%d",sizeof(char*));     /*  prints 8 */
printf("%d",sizeof(short*));    /*  prints 8 */
printf("%d",sizeof(int*));      /*  prints 8 */
printf("%d",sizeof(long*));     /*  prints 8 */
printf("%d",sizeof(float*));    /*  prints 8 */
printf("%d",sizeof(double*));   /*  prints 8 */
```

On a 64-bit computer

A (tortuous) pointer example

```
#include <stdio.h>

int main() {
   int i = 10;
   int** p1;
   int* p2;

   p1 = &p2;
   *p1 = &i;
   *p2 /= 2;

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

   return 0;
}
```



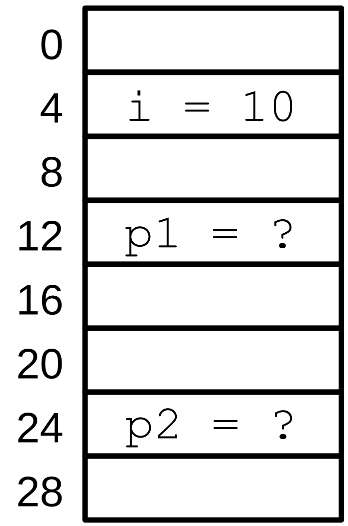
Computer memory

Output:

```
computer:~> ./pointers
```

A (tortuous) pointer example

```
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *p2 /= 2;

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

    return 0;
}
```



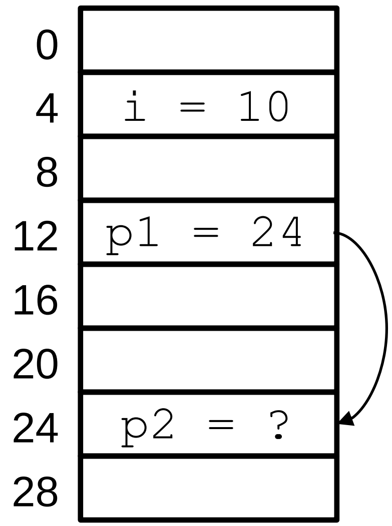
Computer memory

Output:

```
computer:~> ./pointers
```

A (tortuous) pointer example

```c
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *p2 /= 2;

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

    return 0;
}
```



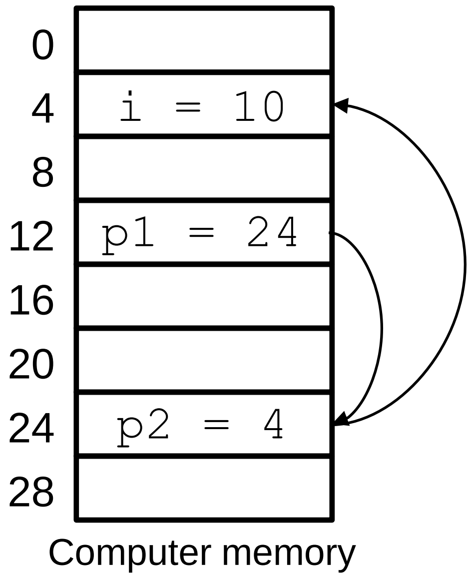
Computer memory

Output:

```
computer:~> ./pointers
```

A (tortuous) pointer example

```c
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *p2 /= 2;

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

    return 0;
}
```

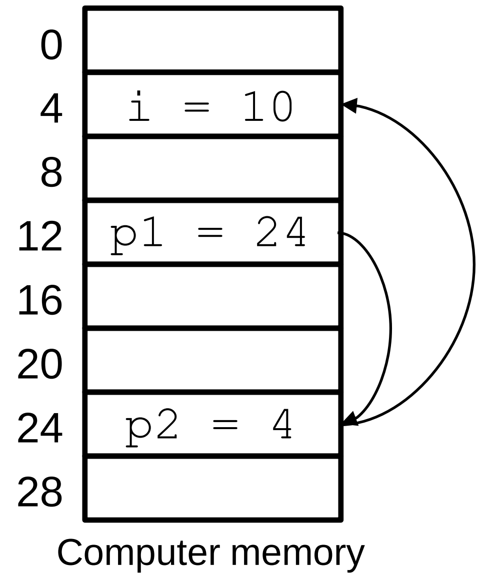


Output:

```
computer:~> ./pointers
```

A (tortuous) pointer example

```
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *p2 /= 2;

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

    return 0;
}
```

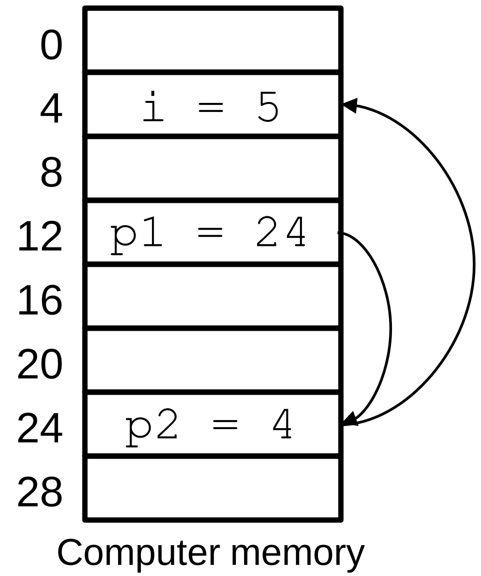


Output:

```
computer:~> ./pointers
```

A (tortuous) pointer example

```
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *p2 /= 2;

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

    return 0;
}
```

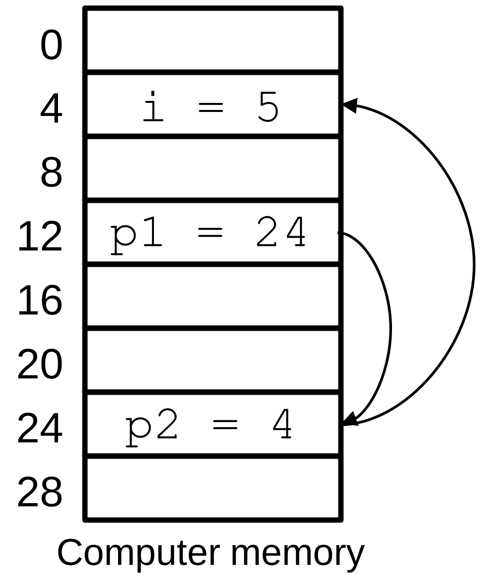


Output:

```
computer:~> ./pointers
```

A (tortuous) pointer example

```c
#include <stdio.h>

int main() {
    int i = 10;
    int** p1;
    int* p2;

    p1 = &p2;
    *p1 = &i;
    *p2 /= 2;

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

    return 0;
}
```

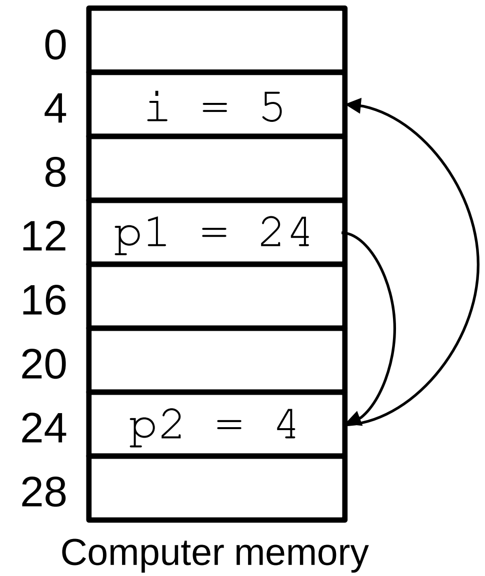


Output:

```
computer:~> ./pointers
```

A (tortuous) pointer example

```
#include <stdio.h>

int main() {
   int i = 10;
   int** p1;
   int* p2;

   p1 = &p2;
   *p1 = &i;
   *p2 /= 2;

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

   return 0;
}
```



Output:

```
computer:~> ./pointers
computer:~> i = 5
```

Dynamic allocation of memory

- MATLAB automatically grows matrices as you continue to add more elements
- These data structures are **dynamical** because they grow automatically in memory as you add data to them.
- In C, you **cannot** do that without managing memory yourself.
- In this code sample, for instance, the array `signal` can contain 50 integers and you cannot make it grow further.
- In many cases, you do not know at **compile time** the size of your data structure. In such cases, you need to **allocate memory dynamically!**

This value has to be a constant!

```
int signal[50];
signal[0] = 0;
signal[1] = 4;
signal[2] = 5;
signal[3] = 4;
signal[4] = 3;
...
```

Dynamic allocation of memory

- To allocate a certain amount of memory, you can use the function `malloc(size)`, where *`size`* is the number of bytes of memory requested (which does not have to be constant).
- `malloc` returns a pointer to the first byte of memory which has been allocated.
- As a result, the static array declaration `int signal[50]` becomes, in its dynamic version:

```
int* signal = malloc(50 * sizeof(int));
signal[0] = 0;
signal[1] = 4;
signal[2] = 5;
signal[3] = 4;
signal[4] = 3;
...
```

This value does not have to be a constant!

Freeing the memory

- If you allocated some memory dynamically, the compiler will **not** take care of freeing the allocated block of memory when you no longer need it.
- Use the function `free(void *ptr)` to make the block available to be allocated again.
- If you perform a `malloc` without its `free` counterpart, you will create a memory leak.
- Therefore, write a `free` for each `malloc` you write.
- After you free memory, you can no longer access it.

```
int* signal = malloc(50 * sizeof(int));
// ...
free(signal);
```

Dynamically allocating memory

```
#include <stdlib.h>

#define MAX_SIZE 1000000

int main() {
  int i;
  int *v; // a vector

  // create a vector of size i
  for (i = 1; i < MAX_SIZE; ++i) {
    v = malloc(i*sizeof(int));
    // do something with vector v
  }

  return 0;
}
```

- Each iteration of the loop, an increasingly larger chunk of memory is allocated with **`malloc`**
- These chunks are never freed, and the program allocates a total of 2,000 GB of memory before terminating!

Dynamically allocating memory

```
#include <stdlib.h>

#define MAX_SIZE 1000000

int main() {
  int i;
  int *v; // a vector

  // create a vector of size i
  for (i = 1; i < MAX_SIZE; ++i) {
    v = malloc(i*sizeof(int));
    // do something with vector v
    free(v); // free memory
  }

  return 0;
}
```

- Each iteration of the loop, an increasingly larger chunk of memory is allocated with **`malloc`**
- These chunks are never freed, and the program allocates a total of 2,000 GB of memory before terminating!
- Calling **`free`** inside the loop means that we never allocate more than 4 MB at a time

Beyond this lecture

- What you learned today are the *basics* of memory management, i.e., **what you need to know as a C programmer**.
- There are further subtleties, which we **do not expect you to understand in depth**, but it is worth knowing that they exist:
  - the ordering of individually addressable units (words, bytes, or even bits) within a longer data word (endianness) might differ from platform to platform
  - memory is actually divided into two parts: (i) the **stack**, on which variables that are declared at **compile time** are stored in order of **decreasing address**; (ii) the **heap**, on which variables that are **dynamically allocated** are stored.
  - there are further types of memory, which you cannot access in C without resourcing to assembler instructions: (i) the **registers**, which are located inside the processor, are extremely fast, but very limited (a few hundreds of bytes); (ii) the **cache**, which is a fast, but small memory (a few megabytes), and is used by the processor to perform "caching" (i.e., pre-fetching and storing chunks of data that are likely to be used or re-used soon).
- Most of these details are **platform-dependent** (and therefore mostly handled by the compiler)

Debugging with gdb

A (tortuous) pointer example

```
#include <stdio.h>

int main() {
  int i = 10;
  int** p1;
  int* p2;

  p1 = &p2;
  *p1 = &i;
  *p2 /= 2;

  return 0;
}
```

What is the value of `i`?

What about now?

Debugging

- Debuggers allow you to step through and examine the effects of your code as it executes
- Many IDEs have a visual debugger built in, but in this class we will use `gdb`, which operates from the command line
- `gdb` has tons of features, but we only need to know a few for it to be an extremely powerful tool

```
$ gcc -g -o pointers pointers.c

$ gdb ./pointers

(gdb) start
```

Basic commands

- Start your program by typing `start` at the `gdb` prompt
- Your program will execute until it reaches a "breakpoint". A breakpoint is automatically inserted at the first line of your main function.
- Breakpoints are added with "`break filename.c:<line>`"
- Execution can be resumed with "`continue`"

Debugging example

```
(gdb)
```

Debugging example

```
(gdb) start
```

Debugging example

```
(gdb) start
Temporary breakpoint 1, main () at pointers.c:4
4           int i = 10;
(gdb)
```

Debugging example

```
(gdb) start
Temporary breakpoint 1, main () at pointers.c:4
4           int i = 10;
(gdb) break pointers.c:10
```

Debugging example

```
(gdb) start
Temporary breakpoint 1, main () at pointers.c:4
4           int i = 10;
(gdb) break pointers.c:10
Breakpoint 2 at 0x4011b8: file pointers.c, line 10.
(gdb)
```

Debugging example

```
(gdb) start
Temporary breakpoint 1, main () at pointers.c:4
4           int i = 10;
(gdb) break pointers.c:10
Breakpoint 2 at 0x4011b8: file pointers.c, line 10.
(gdb) continue
```

Debugging example

```
(gdb) start
Temporary breakpoint 1, main () at pointers.c:4
4          int i = 10;
(gdb) break pointers.c:10
Breakpoint 2 at 0x4011b8: file pointers.c, line 10.
(gdb) continue
Continuing.
Breakpoint 2, main () at pointers.c:10
10 *(&p2[1]-1) /= 2;
(gdb)
```

Inspecting variables

- To inspect the values of different variables, use the "`print`" command

```
Breakpoint 2, main () at pointers.c:10
10 *(&p2[1]-1) /= 2;
(gdb)
```

Inspecting variables

- To inspect the values of different variables, use the "`print`" command

```
Breakpoint 2, main () at pointers.c:10
10 *(&p2[1]-1) /= 2;
(gdb) print &i
```

Inspecting variables

- To inspect the values of different variables, use the "`print`" command

```
Breakpoint 2, main () at pointers.c:10
10 *(&p2[1]-1) /= 2;
(gdb) print &i
$1 = (int *) 0x28abf8
(gdb)
```

Inspecting variables

- To inspect the values of different variables, use the "`print`" command

```
Breakpoint 2, main () at pointers.c:10
10 *(&p2[1]-1) /= 2;
(gdb) print &i
$1 = (int *) 0x28abf8
(gdb) print &p2[1]-1
```

Inspecting variables

- To inspect the values of different variables, use the "`print`" command

```
Breakpoint 2, main () at pointers.c:10
10 *(&p2[1]-1) /= 2;
(gdb) print &i
$1 = (int *) 0x28abf8
(gdb) print &p2[1]-1
$2 = (int *) 0x28abf8
```

Step by step navigation

- Setting a breakpoint on every line of a function would be very tedious!
- Use the `step` and `next` commands to navigate through your code one line at a time
- `step` will enter function calls
- `next` will skip them

```
int main() {
  myfunction(a);
  printf("a = %d\n", a);

  return 0;
}
```

```
void myfunction(int a) {
  // perform calculations
}
```

Step by step navigation

- Setting a breakpoint on every line of a function would be very tedious!
- Use the `step` and `next` commands to navigate through your code one line at a time
- `step` will enter function calls
- `next` will skip them

```
int main() {
  myfunction(a);
  printf("a = %d\n", a);

  return 0;
}
```

```
void myfunction(int a) {
  // perform calculations
}
```

Step by step navigation

- Setting a breakpoint on every line of a function would be very tedious!
- Use the `step` and `next` commands to navigate through your code one line at a time
- `step` will enter function calls
- `next` will skip them

```
int main() {
  myfunction(a);
  printf("a = %d\n", a);

  return 0;
}
```

```
void myfunction(int a) {
  // perform calculations
}
```

Step by step navigation

- Setting a breakpoint on every line of a function would be very tedious!
- Use the `step` and `next` commands to navigate through your code one line at a time
- `step` will enter function calls
- `next` will skip them

```
int main() {
  myfunction(a);
  printf("a = %d\n", a);

  return 0;
}
```

```
void myfunction(int a) {
  // perform calculations
}
```

Step by step navigation

- Setting a breakpoint on every line of a function would be very tedious!
- Use the `step` and `next` commands to navigate through your code one line at a time
- `step` will enter function calls
- `next` will skip them

```
int main() {
  myfunction(a);
  printf("a = %d\n", a);

  return 0;
}
```

```
void myfunction(int a) {
  // perform calculations
}
```

Conclusion

Take-home messages

- A pointer is a variable that contains the address of another variable.
- An array is not a pointer, but acts like one in most cases! Arrays simply address a sequence of values.
- Memory can be either **statically** (at compile time) or **dynamically** (at run time) allocated:
  - **Static allocation** does not require manual deallocation.
  - **Dynamic allocation** requires manual deallocation (using `free`).
- Recall that computer memory has multiple layers of complexity, even though **we do not expect you to know them in details**.
- Debugging with `printf` is still okay, but a debugger like gdb can be more useful in many situations, there is also "valgrind" for memory management.

Additional Literature – Week 6

Programming in C
Stephen G. Koch
ISBN-13: 978-0672326660

C Programming Language
Brian W. Kernighan,
Dennis M. Ritchie
ISBN-13: 978-0131103627

Popular C link
http://www.c-faq.com