# Introduction to C for Java Programmers

A Course in 8 Chapters

FHA SS04



## Table of Contents

# 1. Introduction

- C is a very popular programming language
- C is portable and efficient
- C is a subset of C++
- C and Java are syntactically very similar



# 1 C Program Components

- A C program is a collection of functions (procedures, subroutines),
- can contain global variables and
- can span multiple files.
- A function is a collection of statements,
- encloses code in braces: { }.

# C Program Example

```
/* hello.c */

#include <stdio.h>

int main() {
  puts("Hello World!");
  return 0;
}
```





# Comments and Statements

- Comments are delimited by /* ... */, can span multiple lines.
- Statements contain one or more expressions:
  - Function calls,
  - Numeric operations, etc,
- end with a semi-colon,
- do not have to be on their own line.
- C uses a free-format syntax.

## 1 Include Files

- The **`#include`** directive inserts the text of a file into the compilation stream,
- is used for function declaration and defining constants.
- Standard library files are included with angle brackets: **`<stdio.h>`**
- Any file can be included using quotes: **`#include "myfile.h"`**



## 1 Standard I/O

- Provides console, file and formatted I/O.
- 3 pre-defined I/O streams:
  - **`stdin`** input (keyboard)
  - **`stdout`** output (screen)
  - **`stderr`** error (screen)
- Console functions implicitly use **`stdin`** or **`stdout`**.

# 1 Keyboard Input

```
/* average.c - Average 2 integers */

#include <stdio.h>

int main() {
  // Declarations must be at beginning:
  int num1, num2;
  float sum;

  puts("Enter an integer:");
  scanf("%d", &num1);
  puts("Enter a second integer:");
  scanf("%d", &num2);
  sum = num1 + num2;
  printf("The average is %f\n", sum/2);
  return 0;
}
```



# 1 Sample Output

```
Enter an integer: 10
Enter a second integer: 23
The average is 16.500000
```

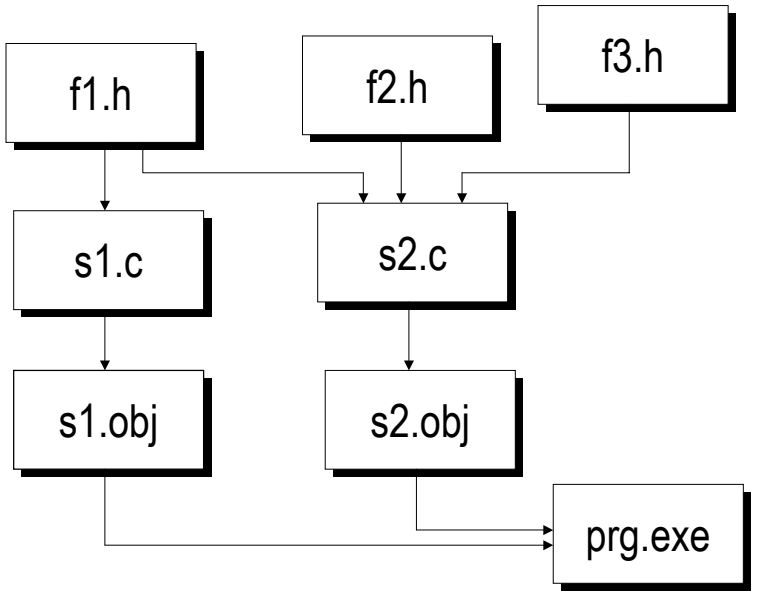
The following statement requests 2-decimal format:

```
printf("Av: %.2f\n", sum/2);
```

```
Av: 16.50
```

1

# C Executable






1

# Summary

- Program source resides in one or more text files
- Source files can **`#include`** one or more header files
- Source files contain one or more functions
- Functions contain statements
- 3 pre-defined I/O streams

# 1 Exercise

Code the Hello World program with the C compiler of your choice using for example:
- emacs/gcc (linux)
- emacs/gcc (cygwin/windows)
- edit/borland c compiler (dos)
- ms visual cpp (windows)
- ...



# Table of Contents

## 2. Fundamental Data Types

- Integers
- Characters
- Floating-point numbers
- Literals and Constants
- Arithmetic Conversions
- Casts



## 2 Integers

- Come in different sizes:
  - Not necessarily distinct!
  - Are signed by default
- **`int`** your machine's word size (at least 16 bits)
- **`short int`** usually the same as **`int`** on 16-bit platforms
- **`long int`** at least 32 bits usually the same as **`int`** on 32-bit platforms

## 2 Characters

- Are really just integers. Character encoding is platform-specific (ASCII, Unicode...)
- **`char`** at least 8 bits
- **`wchar_t`** “wide character ” usually the same as **`unsigned int`** (Unicode)



## 2 Numeric Limits

- Integral limits are in **`<limits.h>`**
- Floating-point limits are in **`<float.h>`**

## 2 Sample Numeric Limits

```
/* limits.c - Illustrate integral limits */
#include <stdio.h>
#include <limits.h>

int main() {
  printf(
    "char:\n [%d, %d]\n", CHAR_MIN, CHAR_MAX);
  printf(
    "short:\n [%d, %d]\n", SHRT_MIN, SHRT_MAX);
  printf(
    "int:\n [%d, %d]\n", INT_MIN, INT_MAX);
  printf(
    "long:\n [%ld, %ld]\n", LONG_MIN, LONG_MAX);
  return 0;
}
```



## 2 Numeric Limits Output

```
char:
 [-128, 127]
short:
 [-32768, 32767]
int:
 [-2147483648, 2147483647]
long:
 [-2147483648, 2147483647]
```

# Floating-Point

- **`float`** "Single precision"
- **`double`** "Double precision" the default
- **`long double`** extended precision could be same as **`double`**





# Sample Floating-Point Limits

```
/* float.c - Illustrate floating-pt. limits */
#include <stdio.h>
#include <float.h>

int main() {
  printf("radix: %d\n\n", FLT_RADIX);
  printf(
    "float: %d radix digits\n", FLT_MANT_DIG);
  printf(" [%g, %g]\n\n", FLT_MIN, FLT_MAX);
  printf(
    "double: %d radix digits\n", DBL_MANT_DIG);
  printf(" [%g, %g]\n\n", DBL_MIN, DBL_MAX);
  printf(
    "long double: %d radix digits\n",
    LDBL_MANT_DIG);
  printf(" [%Lg, %Lg]\n", LDBL_MIN, LDBL_MAX);
  return 0;
}
```

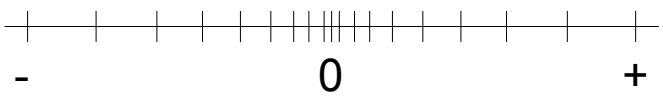
## 2 Numeric Floating-Point Output

```
radix: 2

float: 24 radix digits
 [1.17549e-38, 3.40282e+38]

double: 53 radix digits
 [2.22507e-308, 1.79769e+308]

long double: 64 radix digits
 [3.3621e-4932, 1.18973e+4932]
```



## 2 Non-uniform Distribution

- Dense near zero
- Sparse away from zero



# 2 Sample Missing Numbers

```
/* missing.c */

#include <stdio.h>
#include <limits.h>

main() {
  float x = ULONG_MAX;

  double y = ULONG_MAX;
  long double z = ULONG_MAX;

  printf(
    "%f\n\n%f\n\n%Lf\n", x, y, z);
}
```



# 2 Output Missing Numbers

```
4294967296.000000

4294967295.000000

4294967295.000000
```

## 2 Literals

- `int i = 9, j = 017,`
  `k = 0x7f, l = 0x7F;`
- `char c = 'a', c2 = 97;`
- `long n = 1234567L;`
- `float x = 1.0F;`
- `double y = 2.3;`
- `long double z = 4.5L;`
- `char string[] = "hello";`



## 2 Special Character Literals

- `'\n'` newline
- `'\t'` tab
- `'\0'` null byte (ASCII 0)
- `'\\'` backslash (\)
- `'\b'` backspace
- `'\r'` carriage return
- `'\f'` form feed
- `'\ddd'` octal bit pattern

## 2 Constants

- Variables that cannot be modified
- **`const`** keyword
- Must be initialized during compile-time with a literal:
  **`const int i = 7;`**
- Cannot be used as array dimensions in C
  (But can in C++!)



## 2 Macro Substitution

- Use the **`#define`** directive
- Text substitution is done by the preprocessor
- A way for defining constants
- In C often used as array dimensions:

```
#define SIZE 100
int a[SIZE];
```

## 2 Arithmetic I

- Integer vs. floating-point
- Integer arithmetic truncates any fraction in the result:

```
int i = 2;
int j = 3;
int k = i / j; // k is 0
```



## 2 Arithmetic II

- Floating-point arithmetic suffers from *round-off error:*
  - The result may not be in the set of machine numbers
  - How it rounds is platform-specific

## 2 Promotions

- All integral operations use either **`int`** or **`long`** arithmetic
- A numeric operation assumes the precision of its largest type operand
- Smaller operands are temporarily "widened" ( "promoted") automatically



## 2 Narrowing Conversions

- Beware of narrowing conversions: If the value is not in the range of the smaller type, the result is undefined:

```
char c;
long int l = 123456;
c = l;
```

## 2 Casts

- Allow user-defined explicit conversions
- Precede the expression with the target type in parentheses:

```
int i = (int)x;
```

- Force the precision of an operation, for example:

```
float x = (float)i / j;
```



## 2 Summary

- Built-in C data types are numeric: integer and floating-point
- Integer arithmetic truncates fractions
- Floating-point arithmetic is inexact
- Operands are widened as necessary
- You can force conversions with a cast

## 2 Exercise

Write a program that reads a real number (with a nonzero fractional part) from the keyboard and rounds it to the nearest integer. Print out the original number and the rounded result.



## 2 Solution

```
/* exercise2.c – Round to nearest int */
#include <stdio.h>

int main() {
  float x;
  int n;

  printf("Enter a real number: ");
  fflush(stdout);
  scanf("%f", &x);
  n = (int)(x + 0.5);
  printf("%f rounded == %d\n", x, n);

  return 0;
}
```

# Table of Contents





# 3. Operators

- Mathematical
- Relational
- Logical
- Bitwise
- Assignment
- Operator Associativity and Precedence

## 3 Operator Cardinality

- All operators are either *unary* or *binary*
- Exception:
  The "conditional" operator is *ternary:*

```
max = x > y ? x : y;
```



## 3 Mathematical Operators

- Additive: `+, -, ++, --`

```
i = j++;     // i = j;
             // j = j + 1;
i = ++j;     // j = j + 1;
             // i = j;
```

- Multiplicative: `*, /, %`

```
i = 10 % 3; // = 1
```

## 3 Relational Operators

- Equality: `==`
  ```
  if (i == j) ...;
  ```
  *not*
  ```
  if (i = j) ...; // error!
  ```
- Inequality: `!=`
- Greater-than: `>`, `>=`
- Less-than: `<`, `<=`



## 3 Boolean Expressions

- There is no *boolean* type in C (Java and C++ have them)
- Truth values:
  - Zero is false
  - Non-zero is true

## 3 Logical Operators

- AND: **&&**

```
if (i < n && a[i] == 99)
```

- OR: **||**

```
if (i == 2 || i == 3)
if ((i == 2) || (i == 3))
```

- NOT: **!**

```
if (!(i == 2 || i == 3))
```



## 3 Bitwise Operators

- AND: **&**
- OR: **|**
- XOR: **^**
- **1**'s Complement: **~**
- Shift left: **<<**
- Shift right: **>>**

# 3 Bitwise Example

```
/* bitwise.c - Illustrate bitwise operands */
#include <stdio.h>

int main() {
  short int n = 0x00a4;    // 00000000 10100100
  short int m = 0x00b7;    // 00000000 10110111

  printf("n & m:  %04x\n", n & m);
  printf("n | m:  %04x\n", n | m);
  printf("n ^ m:  %04x\n", n ^ m);
  printf("~n:     %04x\n", ~n);
  printf("n << 3: %04x\n", n << 3);
  printf("n >> 3: %04x\n", n >> 3);
  printf("~n:     %04hx\n", ~n);
  return 0;
}
```



# 3 Output

```
n & m:  00a4     (0000000010100100)
n | m:  00b7     (0000000010110111)
n ^ m:  0013     (0000000000010011)
~n:     ffffff5b (1 ... 1101011011)
n << 3: 0520     (0000010100100000)
n >> 3: 0014     (0000000000010100)
~n:     ff5b     suppressed highest
                 two bytes
```

# Assignment Operators

- Assignment can be combined with other binary operators:
  **+=**, **-=**, ***=**, **/=**, **%=**,
  **>>=**, **<<=**,
  **&=**, **^=**, **|=**

```
i += 5;     // i = i + 5
```





# Operator Associativity

- Governs the order in which adjacent operations execute
- Right-to-left:
  - Unary and assignment operators

```
i = j = k  // i = (j = k)
```

- Left-to-right:
  - Everything else

```
i + j + k  // (i + j) + k
```

3

## Operator Precedence

- Follows mathematical intuition (mostly)
  - Unary, then multiplicative, then additive
- Unary operators are high priority
- Assignment is last (almost, except for the comma op)
- Beware bitwise operators! When in doubt, use parentheses



3

## Summary

- The modulus op (%) gives the remainder from integer division
- The equality op has 2 equal signs (==)
- In boolean contexts, zero is false, non-zero is true
- Unary and assignment ops group right-to-left, all others left-to-right
- Beware the precedence of bitwise operators

# 3 Exercise

Write a program that reads three integers from the keyboard and prints out the sum of all the even numbers and the sum of all the odds.

For example, if the numbers are 1, 2 and 3, the output is:

```
Sum of evens: 2
Sum of odds: 4
```



# 3 Solution

```
/* exercise3.c – Demonstrate operand use. */
#include <stdio.h>

int main() {
  int osum = 0, esum = 0, n;

  printf(" Input first int: ");
  scanf("%d", &n);
  if (n % 2 == 0) esum += n; else osum += n;

  printf(" Input second int: ");
  scanf("%d", &n);
  if (n % 2 == 0) esum += n; else osum += n;

  printf(" Input third int: ");
  scanf("%d", &n);
  if (n % 2 == 0) esum += n; else osum += n;

  printf("Sum of evens: %d\n", esum);
  printf("Sum of odds: %d\n", osum);
  return 0;
}
```

# Table of Contents





# 4. Program Flow

- Structured Programming
- Decision-making
- Repetition
- Branching

# Structured Programming

- In theory, all processes can be expressed via these constructs:
  - Sequences of statements
  - Selection (alternation)
  - Repetition
  - Along with an arbitrary number of boolean flags
- Most languages add some sort of direct branching capability as well (e.g. goto)





# Decision Making

- "Selection"
- **`if-then-else`**
- **`case`** statement
  - Special case of **`if-then-else`**
  - Selects from a set of integers

# 4 The `if` Statement

```
if (<boolean expression>) {
  <statement-if-true>;
}
else {
  <statement-if-false>;
}
```

Compound statements must be enclosed in braces { }



# 4 Example

```
/* age.c – Have your your age commented */
#include <stdio.h>

int main() {
  int age;
  puts("Enter your age: ");
  scanf("%d", &age);
  if (age < 20) {
    puts("youth");
  }
  else if (age < 40) {
    puts("prime");
  }
  // ...
  else {
    printf("Are you really %d?\n", age);
    // ...
  }
  return 0;
}
```

## Dangling `else`

- Omitting braces when using nested **`if`**s and an **`else`** may not result in what you expected
  - Indentation reflecting semantics:

```
// wrong
if (n > 0)
  if (a > b)
    z = a;
else
  z = b;
```

```
// correct
if (n > 0 )
  if (a > b)
    z = a;
  else
    z = b;
```



## The `switch` Statement

- Selects from a set of integral values
- Each case can be delimited by a **`break`**, **o**therwise you fall through to the next case
- Don't define variables inside a **`switch`**
- The optional **`default`** case executes if none are selected

# Example

```
/* age2.c – Demonstrate the use of switch */
#include <stdio.h>

int main() {
  int age;
  puts("Enter your age: ");
  scanf("%d", &age);
  switch (age / 20) {
    case 0:
      puts("youth");
      break;
    case 1:
      puts("prime");
      break;
    // ...
    default:
      printf("Are you really %d?\n", age);
    // ...
  }
  return 0;
}
```



# Repetition

- 3 types of loops:
  - `while (<condition>) { <statement>; }`
  - `do { <statement>; } while (<condition>);`
  - `for (<init>; <cond>; <iter>) { <statement>; }`

## 4 `while` Loop

```
/* Count from 1 to n */
i = 1;
while (i <= n) {
  printf("%d ", i);
  i += 1;
}

/* A shorter version */
i = 1;
while (i <= n)
  printf("%d ", i++);
```



## 4 `do-while` Loop

```
/* Count from 1 to n */

i = 1;

do {
  printf("%d ", i++);
} while (i <= n);
```

## 4 `for` Loop

```
/* Count from 1 to n */

for (i = 1; i <= n; i++) {
  printf("%d ", i);
}
```



## 4 Branching

- **`break`**
  Exits the innermost enclosing loop or **`switch`**
- **`continue`**
  Cycles a loop (i.e., jumps to test)
- **`goto`**
  Jumps to a label
  Useful for exiting nested loops and switches

# 4 Example

```
/* branch.c - Illustrate branching */
/* Find an odd number whose digits add to 7.
   Assume 2 digits only. */

#include <stdio.h>
#define SIZE 6

int main() {
  int i, nums[SIZE] = {10, 21, 25, 32, 43, 54};

  for (i = 0; i < SIZE; ++i) {
    int dig1, dig2;
    if (nums[i] % 2 == 0) {
      continue;          /* skip evens */
    }
    dig2 = nums[i] % 10;
    dig1 = nums[i] / 10;
    if (dig1 + dig2 == 7) {
      printf("found %d\n", nums[i]);
      break;
    }
  }
  return 0;
}
```



# 4 Summary

- All algorithms can be expressed with
  - Simple statements,
  - Decisions,
  - Loops,
  - Plus some flags, maybe.
- Use branching sparingly
- Most loops are **`while`** or **`for`** loops.
  You usually test first.

# 4 Exercise

- Rewrite the odd/even number program from the previous section to process an arbitrary number of input integers. Keep reading until the user enters a 0.
  Use a **`switch`** statement to determine the number's parity.



# 4 Solution

```
/* exercise4.c – Process arbitrary number of
   ints. Keep reading till 0 entered. */
#include <stdio.h>

int main() {
  int n, osum = 0, esum = 0;

  /* scanf returns the number of items read */
  while (scanf("%d", &n) == 1) {
    if (n == 0) break;
    switch (n % 2) {
      case 0: esum += n; break;
      case 1: osum += n; break;
    }
  }
  printf("Sum of evens: %d\n", esum);
  printf("Sum of odds: %d\n", osum);
  return 0;
}
```

# Table of Contents





# 5. Compound Data Types

- Arrays
- Strings
- Structures
- Unions

# 5 Arrays

- The quintessential data structure!
- Homogeneous, fixed-length sequences
- Of any type whatsoever
- Random access via the indexing operator **`[]`**
- Indexing starts at **`0`**, ends at **`n - 1`**



# 5 Example

```
/* reverse.c - Print input sequence backwards */
#include <stdio.h>
#define SIZE 20

int main() {
  int i, n;
  int nums[SIZE];

  /* Read numbers into array. Stop if 0 found */
  for (n = 0; n < SIZE; ++n) {
    int input;
    scanf("%d", &input);
    if (input == 0) break;
    nums[n] = input;
  }

  for (i = n - 1; i >= 0; --i) {
    printf("%d ", nums[i]);
  }
  return 0;
}
```

## 5 Array Initialization

- Can use an *initializer list*:
  ```
  int a[] = {10, 20, 30, 40, 50};
  ```
- The dimension is optional
  - If provided, it must be >= the number of initializers
  - Those remaining are *zero-initialized*
- If omitted, the dimension becomes the number of initializers



## 5 Multi-dimensional Arrays

- Don't really exist in C!
- C, C++ and Java allow "arrays of arrays":
  - Easier to visualize than hyper-tables
- You use one set of brackets for each dimension
- Can be initialized with nested initializer lists

# Example

- Consider the following 3 x 2 (2-dim) array:

  1 2
  3 4
  5 6

- C stores this linearly: 1 2 3 4 5 6
- The compiler interprets it as an array of 3 “arrays of 2 ints”





# Example cont'd

```
/* 2dim.c - Illustrate a 2-dim array */

#include <stdio.h>

int main() {
  int a[][2] = {{1, 2}, {3, 4}, {5, 6}};
  int i, j;

  for (i = 0; i < 3; ++i) {
    for (j = 0; j < 2; ++j) {
      printf("%d ", a[i][j]);
    }
    putchar('\n');
  }
  return 0;
}
```

5

# Strings

- Arrays of characters
- End with a null byte (`'\0'`) by convention
- C++ and Java have better string capabilities
- String literals implicitly provide the null terminator:
  `"hello"` becomes:
  `'h' 'e' 'l' 'l' 'o' '\0'`



5

# Example

```
/* strings.c - Illustrate C strings */

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

int main() {
  char last[] = {'B', 'o', 'n', 'd', '\0'};
  char first[] = "James";

  printf("last: %s\n", last);
  printf("first: %s\n\n", first);
  printf("last has %d chars\n", strlen(last));
  printf("first has %d chars\n", strlen(first));
  return 0;
}
```

# Output

```
last: Bond
first: James

last has 4 chars
first has 5 chars
```





# <string.h>

- Contains a number of text processing functions
- Most assume null-terminated char-arrays
  - **strcpy**, **strcat**, **memcpy**
  - **strcmp**, **memcmp**
  - **strchr**, **memchr**, **strrchr**, **strstr**, **strtok**

# 5 In-core formatting Example

```
/* incore.c - Illustrate sprintf, sscanf */

#include <stdio.h>

int main() {
  int n = 1;
  float x = 2.0;
  char s[] = "hello";
  char string[BUFSIZ];  // BUFSIZ from stdio.h

  sprintf(
    string, "%d %f j%s", n + 1, x + 2, s + 1);
  puts(string);
  sscanf(string, "%d %f %s", &n, &x, s);
  printf("n: %d, x: %f, s: %s\n", n, x, s);

  return 0;
}
```



# 5 Output

```
2 4.000000 jello
n: 2, x: 4.000000, s: jello
```

5

# Structures

- Data record
- Uses the **`struct`** keyword
- Ordered collection of arbitrary variables (“members”)
- Access members via the dot-operator **`'.'`**
- The key to objects and data abstraction!
- Data members are object attributes



5

# Example I

```
/* struct.c - Illustrate structures */
#include <stdio.h>
#include <string.h>

struct Assistant {  /* Ice Hockey Assistant */
  char last[16];    /* 15 + 1 */
  char first[11];   /* 10 + 1 */
  int assists;
};                  /* don’t forget ';' !!! */

int main() {
  struct Assistant a1 = {"Enz", "Mark", 15};
  struct Assistant a2;
  strcpy(a2.last, "De Rosa");
  strcpy(a2.first, "Eugenio");
  a2.assists = a1.assists - 4;

  // cont'd...
```

## 5 Example I cont'd

```
  printf(
    "#1: {%s, %s, %d}\n",
    a1.last, a1.first, a1.assists);
  printf(
    "#2: {%s, %s, %d}\n",
    a2.last, a2.first, a2.assists);

  return 0;
}
```

Output

```
#1: {Enz, Mark, 15}
#2: {De Rosa, Eugenio, 11}
```



## 5 Nested Structures

- **`struct`** members can be any type
- "Hall of Fame" example: shows a **`struct`** within a **`struct`**

# Example II

```
/* struct2.c - Illustrate nested structures */
#include <stdio.h>
#include <string.h>

struct Assistant {
  char last[16];
  char first[11];
  int assists;
  int year;                    /* new member */
};

struct HallOfFame {
  struct Assistant players[10];    /* nested */
  int nPlayers;
};

// cont'd...
```





# Example II cont'd

```
int main() {
  struct HallOfFame hr;
  int i;

  hr.nPlayers = 0;

  /* Insert first player */
  strcpy(hr.players[hr.nPlayers].last, "Ohm");
  strcpy(hr.players[hr.nPlayers].first, "Ted");
  hr.players[hr.nPlayers].assists = 72;
  hr.players[hr.nPlayers++].year = 1925;

  /* Insert next player */
  strcpy(hr.players[hr.nPlayers].last, "Enz");
  strcpy(hr.players[hr.nPlayers].first, "Mark");
  hr.players[hr.nPlayers].assists = 61;
  hr.players[hr.nPlayers++].year = 1961;

  // cont'd...
```

# Example II cont'd

```
   /* Print players contained in hr: */
   for (i = 0; i < hr.nPlayers; ++i) {
     printf(
       "%d: {%s, %s, %d}\n",
       hr.players[i].year,
       hr.players[i].last,
       hr.players[i].first,
       hr.players[i].assists);
   }
   return 0;
}
```

Output

```
1925: {Ohm, Ted, 72}
1961: {Enz, Mark, 61}
```





# Unions

- Unions are similar to structures in many respects
- But less often used
- A union lets you store a value of a type indicated in its declaration
  - keyword **`union`**
  - size of union in bytes is equal to the largest type

5

# Example

```
/* union.c - Illustrate unions */
#include <stdio.h>
#include <string.h>

union Locality {
  int room;            /* room number */
  char branch[21];     /* or branch string */
};

int main() {
  union Locality office[] = {25, 55};
  printf("Room: %d\n", office[0].room);
  printf("Room: %d\n", office[1].room);

  /* Loosing room number: */
  strcpy(office[0].branch, "Geneva");
  printf("Branch: %s\n", office[0].branch);
  return 0;
}
```



5

# Summary

- Arrays start indexing at 0
- Multi-dimensional arrays are “arrays of arrays”
- Strings are arrays of **`char`** and delimited by a null byte
- Structure are collections of data members
- Arrays and structures support brace-delimited initializer lists

# 5 Exercise

- Define an Employee structure that has members last name, first name, title, and salary.
- Write a program that prompts the user for an arbitrary number of Employees, and stores them in an array of Employee. When the user enters an empty string for the last name, print out the list of Employees.



# 5 Solution

```
/* exercise5.c – Employees */
#include <stdio.h>
#include <string.h>

#define MAXEMPS 10
struct Employee {
  char last[16];
  char first[11];
  char title[16];
  int salary;
};


int main() {
  struct Employee emps[10];
  int n, i;

                                    // cont'd
```

# 5 Solution cont'd

```
  for (n = 0; n < MAXEMPS; ++n) {
    printf("Enter last: ");
    fflush(stdout);
    gets(emps[n].last);
    if (strlen(emps[n].last) == 0) break;

    printf("Enter first: ");
    fflush(stdout);
    gets(emps[n].first);

    printf("Enter title: ");
    fflush(stdout);
    gets(emps[n].title);

    printf("Enter salary: ");
    fflush(stdout);
    scanf("%d", &emps[n].salary);
    getchar();      /* eat newline */
  }                                  // cont'd
```



# 5 Solution cont'd

```
  // Output:
  for (i = 0; i < n; ++i) {
    printf(
      "{%s,%s,%s,%d}\n",
      emps[i].last,
      emps[i].first,
      emps[i].title,
      emps[i].salary);
  }

  return 0;
}
```

# Table of Contents





# 6. Programming with Functions

- Procedural Programming
- Value Semantics
- Function Prototypes
- Scope
- Automatic, Static and Global Variables

# Functions in C

- A collection of statements
  - Defined at file scope
  - Each has a unique name
  - Enclosed in braces: { }
  - Performs some well-defined operation
  - May take arguments
  - May return a value





# Example

```
/* fun1.c - Illustrate a C function */

#include <stdio.h>

float avg(int n, int m) {
  return (n + m) / 2.0;
}


int main() {
  int x, y;

  puts("Enter the first number:");
  scanf("%d", &x);
  puts("Enter the second number:");
  scanf("%d", &y);
  printf("\nThe average is %.2f\n", avg(x, y));
  return 0;
}
```

# Output

```
Enter the 1st number:
11
Enter the 2nd number:
12

The average is 11.50
```





# Value Semantics

- Arguments are passed by value
  - Each formal parameter gets a *copy* of its argument's value
  - The calling argument is not affected
  - The result is returned by value
  - The calling expression gets a temporary copy:

```
a = b + avg(c, d);
```

# The **`void`** keyword

- Used to indicate the absence of a return value
  - i.e., the function is a *procedure:*

```
void f(int x, float y) { … }
```

- Or to disallow arguments:

```
void title(void) {
  printf("Welcome to …");
}
…
title();
```





# Example

```
/* fun2.c - Show return with void. */
/* Also illustrate an array parameter */
#include <stdio.h>

void printInts(int nums[], int n) {
  int i;
  if (n <= 0) return;
  for (i = 0; i < n; ++i) {
    if (i > 0) putchar(',');
    printf("%d", nums[i]);
  }
}

int main() {
  int a[] = {9, 0, 2, 1, 0};
  printInts(a, 5);
  return 0;
}
```

# Using Functions

- The number and types of the calling arguments should match a function's formal parameters
- The compiler can detect usage errors at compile time
  - But you have to provide the information
- Guideline: either *define* or *declare* a function before its first use





# Function Prototypes

- A function declaration
  - Signature (name + param types)
  - Return type
- Lets you define a function in a separate file from where it's called
  - The basis for reusable libraries

# Example

```
/* fun3.c - Illustrate a function prototype */
#include <stdio.h>

float avg(int, int);    /* Prototype */

int main() {
  int x, y;
  puts("Enter the first number:");
  scanf("%d", &x);
  puts("Enter the second number:");
  scanf("%d", &y);
  printf("The average is %.2f\n", avg(x, y));
  return 0;
}

float avg(int n, int m) {
  return (n + m) / 2.0;
}
```



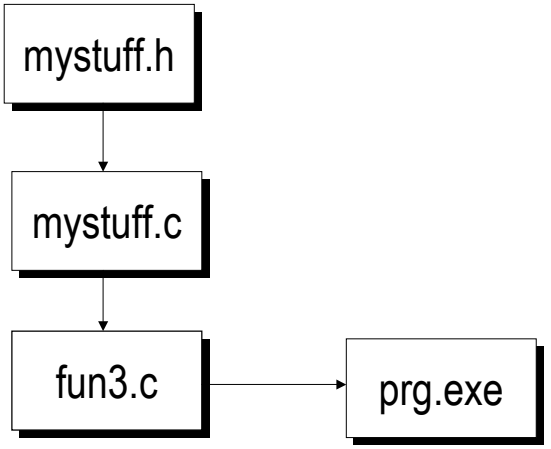


# Modified Example

```
/* mystuff.h */
float avg(int, int);    /* Prototype */
…


/* mystuff.c */
float avg(int n, int m) {
  return (n + m) / 2.0;
}
…


/* fun3.c */
#include <stdio.h>
#include "mystuff.h"

int main() {
  …
  … avg(x, y) …
  return 0;
}
```

6

# Modified Example Executable






6

# Scope

- Where an identifier is visible
- Three basic types:
  - Local (or “block”) scope
  - File scope
  - Program scope
  - (There are others)

# Local Scope

- Within a block, i.e. a set of braces
  - Functions
  - Loops and if-statements
- Visible from the point of declaration until the end of the block





# File Scope

- The region outside of any function
- Visible from declaration to end of file

# 6 Example

```
/* scope.c */

#include <stdio.h>

int i = 3;     /* A "global" variable */

int main() {
  int j;
  printf("%d\n", i);
  for (j = 0; j < i; ++j) {
    int i = 99;
    printf("%d\n", i);
    {
      int i = j;
      printf(" >%d\n", i);
    }
  }
  return 0;
}
```



# 6 Output

```
3
99
 >0
99
 >1
99
 >2
```

6

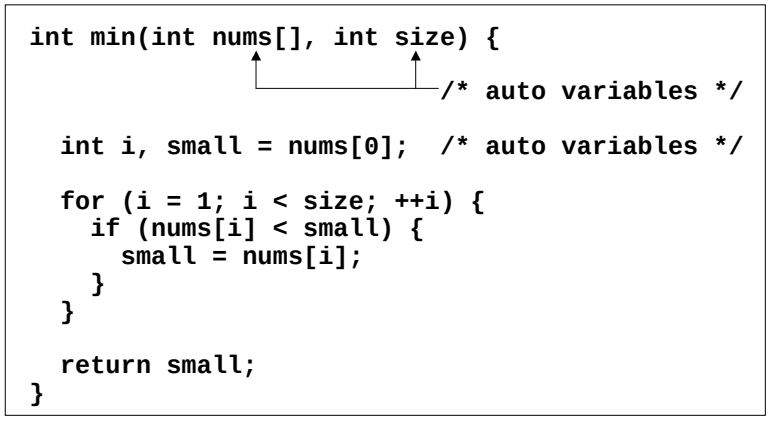
# Automatic Variables

- Allocated on the program stack
- Any variable defined in a block *without* the **`static`** keyword
- Reinitialized each time execution enters its block



6

# Example



# Static Variables

- Reside in a special data area
  - (Static) data segment
- Any variable defined outside a block
  - “File scope”
- Any variable declared inside a block with the **`static`** keyword
- Initialized only once
  - At program startup





# Example

```
/* static.c */

#include <stdio.h>

int count() {
  static int n = 0;
  return ++n;
}


int main() {
  int i;

  for (i = 0; i < 5; ++i) {
    printf("%d ", count());
  }

  return 0;
}
```

## 6 Output

```
1 2 3 4 5
```



## 6 Global Variables

- Have “program scope”
- Accessible outside their source file
- Defined at file scope *without* the **`static`** keyword
- Use the **`extern`** keyword to refer to global variables in other files
- **`static`** + file scope = private to its file

# Example

```
/* file1.c */
int i = 10;           /* global */
static int j = 20;    /* private */

int getJ(void) {
  return j;
}


/* file2.c */
#include <stdio.h>

int main() {
  extern int i;

  /* extern optional for functions: */
  int getJ(void);

  printf("i == %d\n", ++i);      /* i: 11 */
  printf("j == %d\n", getJ());   /* j: 20 */
  return 0;
}
```





# Information Hiding

- Using file **`static`**s in C
- Protects data
- Separates interface from implementation
- Fundamental principle of object-oriented programming
- C++ and Java support it much better

# Stack Example

- **`stack.h`**
  - User function declarations
- **`stack.c`**
  - Function implementation
  - Private definitions
- **`tstack.c`**
  - A test program





# Example

```
/* stack.h - Declarations for a stack of ints */

#define STK_ERROR –32767

void stkPush(int);
int  stkPop(void);
int  stkTop(void);
int  stkSize(void);
int  stkError(void);
```

# Example cont'd

```
/* stack.c - Implementation */
#include "stack.h"

/* Private data: */
#define MAX 10          /* stack limit */
static int error = 0;   /* error flag */
static int data[MAX];   /* the stack */
static int ptr;         /* stack pointer */

/* Function implementation */
void stkPush(int x) {
  if (ptr < MAX) {
    data[ptr++] = x;
    error = 0;
  }
  else {
    error = 1;
  }
}                          // stack.c cont'd...
```





# Example cont'd

```
                           // ...stack.c cont'd
int stkPop(void) {
  if (ptr > 0) {
    int x = data[--ptr];
    error = 0;
    return x;
  }
  else {
    error = 1;
    return STK_ERROR;
  }
}


int stkSize(void) {
  return ptr;
}                          // stack.c cont'd...
```

# Example cont'd

```
                              // ...stack.c cont'd
int stkTop(void) {
  if (ptr > 0) {
    error = 0;
    return data[ptr - 1];
  }
  else {
    error = 1;
    return STK_ERROR;
  }
}

int stkError(void) {
  return error;
}
```





# Example cont'd

```
/* tstack.c - Test the stack of ints */
#include "stack.h"
#include <stdio.h>

int main() {
  int i;

  /* Populate stack */
  for (i = 0; i < 11; ++i) stkPush(i);
  if (stkError()) puts("stack error");
  printf(
    "the last int pushed was %d\n", stkTop());

  /* Pop/print stack */
  while (stkSize() > 0) printf("%d ", stkPop());
  putchar('\n');
  if (!stk_error()) puts("no stack error");
  return 0;
}
```

## 6 Output

```
stack error
the last element pushed was 9
9 8 7 6 5 4 3 2 1 0
no stack error
```



## 6 Summary

- A function is a named collection of statements
  - May take arguments
  - May return a value
- C functions use value semantics
- Scope is a variable's visible region
  - Local
  - File
  - Program
- Hide variables with file **`static`**s

6

# Exercise

- **`exercise5.c`** is split into 3 files: **`employee.h`**, **`employee.c`**, **`exercise6.c`** (similar to the stack example).
- Provide **`employee.c`**, which will contain the **`Employee`** structure definition, private data and function implementations.
- **`employee.h`** and **`exercise6.c`** cp next slides.



6

# Exercise cont'd

```
/* employee.h */
/* Read each field from standard input into
 * the next available Employee slot, as in the
 * exercise5.c. Return the index of the
 * Employee just added, or -1 if array is full.
 */
int addEmployee(void);


/* Return the index of the Employee just
 * printed, or -1 if the index i is invalid.
 */
int printEmployee(int i);


/* Get number of Employees.
 */
int numEmployees(void);
```

# 6 Exercise cont'd

```
/* exercise6.c */
#include "employee.h”
#include <stdio.h>

int main() {
  int i;

  /* Fill Employee array: */
  while (addEmployee() != -1)
    ;

  /* Print each Employee: */
  for (i = 0; i < numEmployees(); ++i) {
    printEmployee(i);
    putchar('\n');
  }

  return 0; }
```



# 6 Solution

```
/* employee.c */
#include "employee.h"
#include <stdio.h>
#include <string.h>

#define MAXEMPS 5

struct Employee {
  char last[16];
  char first[11];
  char title[16];
  int salary;
};

static struct Employee emps[MAXEMPS];
static int nemps = 0;

                                        // cont'd
```

# 6 Solution cont'd

```
int addEmployee(void) {
  if (nemps == MAXEMPS)
    return -1;
  printf("Enter last: ");
  fflush(stdout);
  gets(emps[nemps].last);
  if (strlen(emps[nemps].last) == 0)
    return -1;
  printf("Enter first: ");
  fflush(stdout);
  gets(emps[nemps].first);
  printf("Enter title: ");
  fflush(stdout);
  gets(emps[nemps].title);
  printf("Enter salary: ");
  fflush(stdout);
  scanf("%d", &emps[nemps].salary);
  getchar();     /* eat newline */
  return nemps++;
}                                    // cont'd
```



# 6 Solution cont'd

```
int printEmployee(int i) {
  if (i < 0 || i >= nemps)
    return -1;
  printf(
    "{%s, %s, %s, %d}",
    emps[i].last,
    emps[i].first,
    emps[i].title,
    emps[i].salary);

  return i;
}


int numEmployees(void) {
  return nemps;
}
```

# Table of Contents





# 7. Pointers

- Indirection
- Simulating Call-by-reference
- Pointer and Arrays
- Pointer Arithmetics
- Pointer types
- Command-line Arguments
- Heap variables

7

# Pointers

- A pointer holds a number
  - Interpreted as the address of another “object”
- Declared with its associated type:
  - “pointer to integer”
  - “pointer to character” etc.
- Useful for:
  - Dynamic objects (allocated on the heap)
  - For direct machine access (such as mapped memory)



7

# Pointers Indirection

- Accessing an object through a pointer is called *indirection*
- The “address-of” operator (**&**) obtains an object’s address
- The “de-referencing” operator (*****) refers to the object pointed at

# Example

```
/* indirection.c - Illustrate indirection */
#include <stdio.h>

int main() {
  int i = 7;
  int *ip;

  ip = &i;   /* assign address of i to ip: */
             /* ip now points to i        */
  printf("Address %p contains %d\n", ip, *ip);

  *ip = 8;
  printf("Address %p contains %d\n", ip, *ip);

  return 0;
}
```

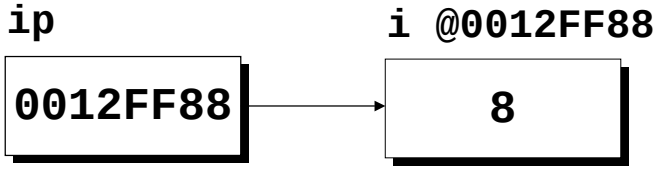




# Output

```
Address 0012FF88 contains 7
Address 0012FF88 contains 8
```

## Pointer Diagram



# Example II

```
/* association.c - Illustrate associated types */
#include <stdio.h>

int main() {
  char str[] = "String";
  char c;
  char *cp = &c;

  double d = 27.9;
  double *dp = &d;

  *cp = str[3];
  printf("c: %c, *cp: %c\n", c, *cp);

  d += 2.2;
  printf("d: %.2f, *dp: %.2f\n", d, *dp);

  return 0;
}
```

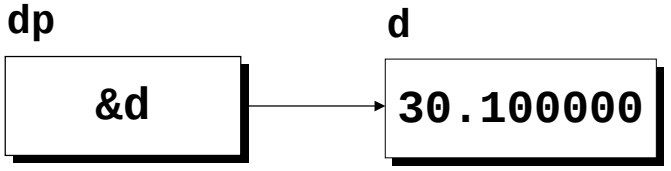




# Output

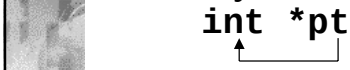
```
c: i, *cp: i
d: 30.10, *dp: 30.10
```

## Pointer Diagram



# Syntax of Declarators

- The syntax of declarators mimic the syntax of use.
  - Symmetry:



```
int *ptr;   /* pointer to int */

*ptr;       /* type is int    */
```

  - Second example:

```
int *(*x)[4];
*(*x)[2];    /* type is int   */
```





# The **NULL** Pointer

- A special value
  - The address 0
- Doesn't point anywhere
- Can be used to compare to other pointers
  - e.g. as a sentinel
  - Returned by selected library functions
- You can't de-reference **NULL**

## The `NULL` Pointer cont'd

- Defined in **`<stdlib.h>`**:
  **`#define NULL ((void *)0)`**
- Assigning:
  **`double *dp = NULL;`**
- Testing:
  **`if (dp != NULL)`**
  In shorthand notation:
  **`if (dp)`**





## Reference Semantics

- Where a formal parameter is just an alias for the calling argument
  - Changing the parameter changes the original argument
- Not supported directly in C
  - Nor in Java
  - But C++ supports it
- In C we fake it with pointers

# Example

```
/* swap.c – Simulate reference semantics */
#include <stdio.h>

void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}


int main() {
  int i = 1, j = 2;

  swap(&i, &j);
  printf("i: %d, j: %d\n", i, j);
  return 0;
}
```





# Pointers and Arrays

- The name of an array represents the address of the first element
  - These two lines are identical:

```
scanf("%s", text);
scanf("%s", &text[0]);
```

- Pointer points to array:

```
char text[] = "Text";
char *cp = text;
```

# Example

```
/* arrAndPtr.c – Illustrate similarity */
#include <stdio.h>

void write1(char text[]) {
  int x = 0;
  while (text[x]) putchar(text[x++]);
}


void write2(char *text) {
  while (*text) putchar(*text++);
}


int main() {
  char text[] = "Text";

  write1(text);
  write2(text);
  return 0;
}
```





# Pointer Arithmetics

- Pointers are incremented due to the type they are pointing to

```
char c, text[] = "Text";
char *cp = text;
cp++;
c = *(cp - 1);
```

- Pointer arithmetics are always type-dependent
- Be careful with precedence

# Example I

```
/* ptrArithm.c – Illustrate ptr arithmetics */
#include <stdio.h>

int main() {
  char text[] = "Text";
  char *cp = text;

  printf(
    "Third element is char %c\n", text[2]);

 printf(
    "Third element is char %c\n", *(text + 2));

 printf(
    "Third element is char %c\n", *(cp + 2));

  return 0;
}
```





# Example II

```
/* ptrArithm2.c – Illustrate ptr arithmetics */
#include <stdio.h>

int main() {
  int fibo[] = {1, 1, 2, 3, 5, 8, 13};

  printf(
    "Third element is int %i\n", fibo[2]);

  printf(
    "Third element is int %i\n", *(fibo + 2));

  return 0;
}
```

# Example III

```
/* precedence.c – Illustrate precedence */
#include <stdio.h>

int main() {
  int fibo[] = {1, 1, 2, 3, 5, 8, 13};

  printf("*(fibo + 3): %i\n", *(fibo + 3));
  printf("(*fibo) + 3: %i\n", (*fibo) + 3);

  return 0;
}
```





# Pointer Types

- Pointer types are referred to as object pointers or function pointers
- A value of a pointer type is
  - the address of an object, or
  - the address of a function returning a type:

    **`int (*fp)();`** pointer to function returning an **`int`**

# Example I

```
/* ptrTofun.c – Illustrate function pointers */
#include <stdio.h>

int echo(int i) {
  return i;
}


int main() {
  int (*fp)();      /* pointer to function   */
                    /* returning an int      */
  fp = echo;
  printf("fp(234): %i\n", fp(234));

  return 0;
}
```





# Example II

```
/* ptrTofun2.c – Show more complex situation */
#include <stdio.h>

int echo(int i) { return i; }
int dummy() { return 0; }


int main() {
  /* array of pointers to functions */
  /* returning integers:            */
  int (*arr[])() = {echo, dummy};

  printf("(*arr[0])(77): %i\n", (*arr[0])(77));
  printf("(*arr[1])(): %i\n", (*arr[1])());

  return 0;
}
```

7

# Generic Object Pointers

- A generic object pointer can be converted to any pointer and vice versa
- Have to be casted to the proper type before de-referencing
- Keyword **`void *`**
- There is no generic function pointer

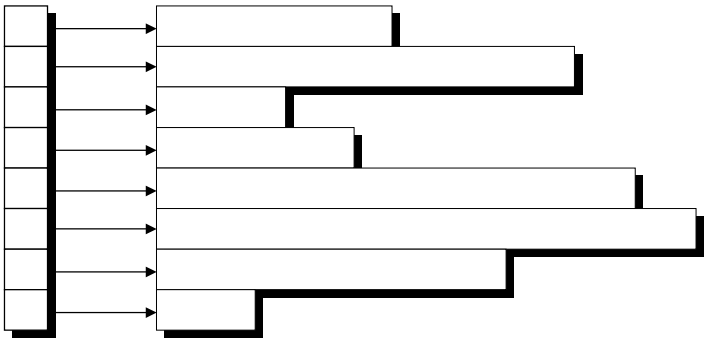


7

# Samples

```
void *genericPtr;
int *intPtr;
char *charPtr;

genericPtr = intPtr;     /* OK */
intPtr = genericPtr;     /* OK */
intPtr = charPtr;        /* Invalid in ISO C */
intPtr = (int *)charPtr; /* OK */


/* Use within function prototypes */
/* memcpy can take any kind of pointer: */
void *memcpy(
  void *dest, const void *src, size_t n);
```

# Ragged Arrays







# Command-line Arguments

- Optional arguments to main

```
int main(int argc, char *argv[]) {
  …
}
```

- **`argv`** is a ragged array

7

# Example

```
/* echo.c - Echo command-line args */
#include <stdio.h>

int main(int argc, char *argv[]) {
  int i;

  for (i = 0; i < argc; ++i) {
    puts(argv[i]);
  }
  return 0;
}
```



7

# Command-line Input

```
C:\sub>.\echo hello there
```

## Output

```
C:\sub\echo.exe
hello
there
```

7

# Heap Variables

- Accessed indirectly through a pointer
  - Returned by **`malloc()`**
- Reside in a unique data area
  - Often called the “heap” or “dynamic storage”
- You give the memory back when done
  - Via **`free()`**



7

# Heap Variables cont'd

- Useful when you don’t know everything ahead of time
  - Like how big an array needs to be

# C Heap Functions

- Defined in **<stdlib.h>**:

```
void *malloc(size_t size);

void free(void *ptr);

void *calloc(
  size_t nelems,
  size_t elem_size);

void *realloc(
  void *ptr, size_t size);
```





# Example

```
/* reverse2.c - Print lines in reverse order
   from input */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXWIDTH 81
#define MAXLINES 100

int main() {
  char *lines[MAXLINES];
  char line[MAXWIDTH];
  int i, n;


                                  // cont'd...
```

# Example cont'd

```
                                        // ...cont'd
 /* Store in a ragged array */
  for (n = 0;
       n < MAXLINES && gets(line) != NULL; ++n)
  {
    if ((lines[n] = malloc(strlen(line) + 1))
         == NULL)
    {
      exit(1);
    }
    strcpy(lines[n], line);
  }

  /* Print in reverse order */
  for (i = 0; i < n; ++i) {
    puts(lines[n – i - 1]);
    free(lines[n – i - 1]);
  }

  return 0;
}
```





# The `sizeof` operator

- Gives the size of a variable or type in bytes
- A compile-time operator
- Array size idiom:
  `int n = sizeof a / sizeof a[0];`
- Must use parentheses with types:
  `float *p = malloc(sizeof(float));`

## `struct`s on the Heap

- Use **`sizeof`** as usual:

```
typedef struct _Employee Employee;
Employee *p =
  malloc(sizeof(Employee));
```

- Accessing members uses a messy syntax:

```
(*p).age = 47;
```

- Alternate syntax, operator (**`->`**):

```
p->age = 47;
```





## `struct`s as Arguments

- You usually pass a **`struct`**'s address
  - Into a pointer parameter
  - Saves time and space
- Equivalent to Java semantics
  - Java never passes objects by value
  - Passes a pointer instead
  - But it's all invisible to you

# Example

```
/* structarg.c - Pass a struct by address */
#include <stdio.h>

struct _Date {
  int year;
  int month;
  int day;
};

typedef struct _Date Date;


void printDate(Date *p) {
  printf(
    "%2d.%2d.%02d", p->day, p->month, p->year);
}

int main() {
  Date d = {2001, 12, 31};
  printDate(&d);
  return 0;
}
```





# Output

```
31.12.2001
```

## Summary

- Pointers allow for indirection
- Pointers to objects and functions
- The **`NULL`** pointer
- Generic pointer **`void *`**
- Array Elements can be referenced by pointers
- Allocation on the heap and pointers
- Pointers and **`struct`**s





## Exercise

- **`employee2.h`** and the test file **`exercise7.c`** are given. The test program creates **`Employee`**s and exercises the various functions declared in **`employee2.h`**.
- Provide the implementation for these functions in a file named **`employee2.c`**:

7

# Exercise cont'd

- **`createEmployee(…)`** allocates an **`struct _Employee`** on the heap and initializes it with its arguments and returns the pointer provided by **`malloc( )`**.



7

# Exercise cont'd

```
/* employee2.h */

#ifndef EMPLOYEE_H
#define EMPLOYEE_H 1


struct _Employee {
  char last[16];
  char first[11];
  char title[16];
  int salary;
};

typedef struct _Employee Employee;

                                    // cont'd...
```

# Exercise cont'd

```
                          // ...employee2.h cont'd

Employee *createEmployee(
  char *, char *, char *, int);
char *getLast(Employee *);
char *getFirst(Employee *);
char *getTitle(Employee *);
int getSalary(Employee *);
void setLast(Employee *, char *);
void setFirst(Employee *, char *);
void setTitle(Employee *, char *);
void setSalary(Employee *, int);
void printEmployee(Employee *);

#endif
```





# Exercise cont'd

```
/* exercise7.c */
#include "employee2.h"
#include <stdio.h>
#include <stdlib.h>

#define MAXEMPS 5

int main() {
  Employee *emps[MAXEMPS];
  Employee *p;
  int i, nEmps = 0;

  emps[nEmps++] =
    createEmployee("Ilg", "Edi", "Dr", 100);
  emps[nEmps++] =
    createEmployee("Rot", "Emma", "Prof", 110);
  if (emps[nEmps-1]->salary != 115) {
    emps[nEmps-1]->salary = 115;
  }
                                    // cont'd...
```

# 7 Exercise cont'd

```
                        // ...exercise7.c cont'd

  p = createEmployee("", "", "", 0);
  setLast(p, "Viola");
  setFirst(p, "Aldo");
  setTitle(p, "Ing FHA");
  setSalary(p, 100);
  emps[nEmps++] = p;

  for (i = 0; i < nEmps; ++i) {
    printEmployee(emps[i]);
    putchar('\n');
    free(emps[i]);
  }

  return 0;
}
```



# 7 Output

```
{Ilg, Edi, Dr, 100}
{Rot, Emma, Prof, 110}
{Viola, Aldo, Ing FHA, 100}
```

# Solution

```
/* employee2.c */
#include "employee2.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

Employee *createEmployee(char *last,
  char *first, char *title, int salary) {
  Employee *p = malloc(sizeof(Employee));
  if (p != NULL) {
    strcpy(p->last, last);
    strcpy(p->first, first);
    strcpy(p->title, title);
    p->salary = salary;
  }
  return p;
}
                                    // cont'd...
```





# Solution cont'd

```
                        // ...employee2.c cont'd
char *getLast(Employee *p) {
  return p ? p->last : "";
}

char *getFirst(Employee *p) {
  return p ? p->first : "";
}

char *getTitle(Employee *p) {
  return p ? p->title : "";
}

int getSalary(Employee *p) {
  return p ? p->salary : 0;
}
                                    // cont'd...
```

# Solution cont'd

```
                        // ...employee2.c cont'd
void setLast(Employee *p, char *last) {
  if (p) strcpy(p->last, last);
}

void setFirst(Employee *p, char *first) {
  if (p) strcpy(p->first, first);
}

void setTitle(Employee *p, char *title) {
  if (p) strcpy(p->title, title);
}

void setSalary(struct Employee *p, int salary)
{
  if (p) p->salary = salary;
}
                                      // cont'd...
```





# Solution cont'd

```
                        // ...employee2.c cont'd
void printEmployee(Employee *p) {
  putchar('{');
  if (p) {
    printf("%s, %s, %s, %d",
    p->last, p->first, p->title, p->salary);
  }
  putchar('}');
}
```

# Table of Contents





# 8. Using Files

- Opening and Closing a file
- Creating a file
- File existence
- Reading a file
- Writing a file
- More file operations

# C and file operations

- C has a vast number of file functions.
- They are defined in <stdio.h>.
- When dealing with files you use file pointers defined in <stdio.h>: **`FILE *fp;`**
- Data of type **`FILE`** store information of an opened file.





# Opening and Closing

- To open a file use **`fopen`**.
- If opening fails **`NULL`** is returned.
- If opened with success preserve the returned file pointer for later use.
- A file has to be "opened" in one of the following modes:
  **`"r"`** read only
  **`"w"`** write / overwrite / create
  **`"a"`** append or create

# 8 Example

```
/* openClose.c - Open and close a file */
#include <stdio.h>

int main() {
  FILE *fp;
  // "r": read only, also write protected files
  fp = fopen("openClose.c", "r");
  if (fp != NULL) {
    flose(fp);
  }

  return 0;
}
```



# 8 fopen fclose

Prototypes in stdio.h:

```
FILE *fopen(
  char *filename, char *mode);

void fclose(FILE *fp);
```

# Creating a file

- "Open" the file with **`fopen`** in mode **`"w"`**.
- Beware: An existent file will be overwritten.
- A new empty file is opened.
- Always test if successful.
- Save the returned file pointer.





# File existence

- There is no function for this purpose in the standard library.
- Trick: Try to read the file.
  Cp. example.

# 8 Example

```
/* fexists.c - Check whether file exists */
#include <stdio.h>
#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

int fexists(char *filename) {
  FILE *fp;
  fp = fopen(filename, "r");
  if (fp != NULL) {
    fclose(fp);
    return TRUE;
  }

  return FALSE;
}
```



# 8 Reading a file

- The end of file (EOF) can be determined with **`int feof(FILE *fp)`**.
  Return value
  0: end not reached,
  ≠0: eof
- A character is read by means of **`int fgetc(FILE *fp)`**. **`fp`** is incremented automatically.

# 8 Example

```
/* readFile.c - Read this file to stdout */
#include <stdio.h>

int main() {
  FILE *fp;

  fp = fopen("readFile.c", "r");
  if (fp != NULL) {
    char c = fgetc(fp);
    while(!feof(fp)) {
      putchar(c);
      c = fgetc(fp);
    }
    fclose(fp);
    return 0;
  }

  return 1;
}
```



# 8 Writing a file

```
/* writeFile.c - Write into a file */
#include <stdio.h>
#include <string.h>

int main() {
  FILE *fp;
  char line[81];
  fp = fopen("text.txt", "w");
  if (fp != NULL) {
    puts("Enter chars, exit with empty line");
    gets(line);
    while(strlen(line) > 0) {
      fprintf(fp, "%s\n", line;
      gets(line);;
    }
    fclose(fp);
    return 0;
  }
  return 1;
}
```

## 8 More file operations

freopen
fseek
fgetc
fgets
fputc
remove
...



## 8 Summary

- A large number of file functions are provided in <stdio.h>.
- File handling uses pointers.
- Always test if file operation was successful. Take appropriate measures if not.

# 8 Exercise

- Write a program which compares the contents of two files:
  - Allow input of the file names via the command line.
  - Count the number of chars while comparing.
  - Exit if there is a difference and report where the difference was found.