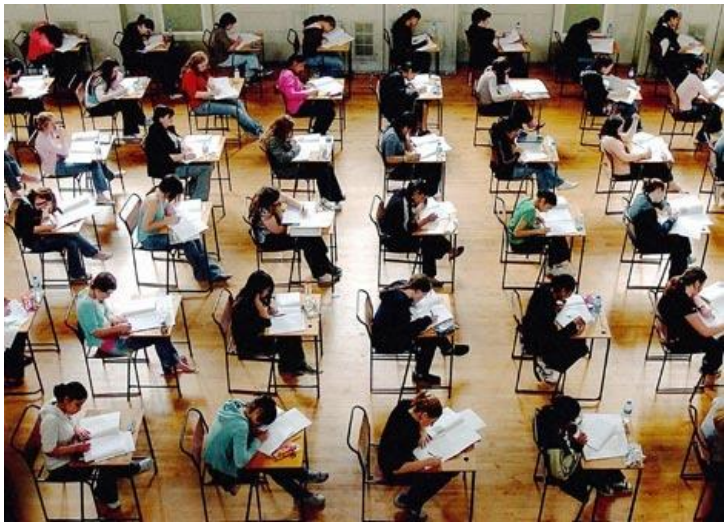
UNIVERSITY of LOUISVILLE
dare to be great

Lecture Eleven
Review of Chapters 1-9

CECS130
Introduction to Programming Languages
Dr. Roman V. Yampolskiy

Midterm 1

- Covers C-Book chapters 1-9 (Lectures 1-9)
- Labs 1-5
- Closed notes/books
- Worth 15% of your final grade



Chapter 1
Creating a Computer Program

1. Write the program in programming language using a text editor. Save the program in a file. This file contains what's known as the **source code.**
2. Compile the program using a special program known as a **compiler**. A compiler converts a source code into a sequence of 0's and 1's.
3. A file containing 0's and 1's is generated. This file is referred to as the **executable file**. "Run" the executable file.

What Does a Program Look Like?

```
/*
This is a simple C Program.
*/
// This is a comment

#include <stdio.h>

main()
{

        printf("\nC you later\n");

}
```

Escape Sequence Description

- **Used to format output**
  - **\' Single quote**
  - **\" Double quote**
  - **\\ Backslash**
  - **\n Newline**
  - **\r Carriage return**
  - **\t Horizontal tab**

Chapter 2
Variable Naming

- A variable may be defined using
  - any uppercase or lowercase character
  - a numerical digit (0 through 9)
  - the underscore character (_).
  - The first character of the variable may not be a numerical digit or underscore.
  - Variable names are case sensitive.
  - Decimal points, commas or any other symbols are not allowed.

Keywords

Keywords are reserved and cannot be used as variable names.

Keywords are lower case.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Constants

- C has a qualifier that tells the application that the value of the variable cannot be changed.
- This qualifier is ***const*** and is used as in:

```
const int myAge = 29;
/* The value of myAge will stay constant at 29 all the time */
```

Operator Precedence Table

	Description	Represented By
1	Parenthesis	() []
1	Structure Access	. ->
2	Unary	! ~ ++ -- - * &
3	Mutiply, Divide, Modulus	* / %
4	Add, Subtract	+ -
5	Shift Right, Left	>> <<
6	Greater, Less Than, etc	> < =
7	Equal, Not Equal	== !=
8	Bitwise AND	&
9	Bitwise Exclusive OR	^
10	Bitwise OR	|
11	Logical AND	&&
12	Logical OR	||
13	Conditional Expression	?:
14	Assignment	= += -= etc
15	Comma	,

prinft/scanf Conversion Codes

- A `%<conversion code>` in the printf/scanf string is replaced by the respective variable.
  - `%c` – a character
  - `%d` – an integer, `%u` – an unsigned integer.
  - `%f` – a float
  - `%lf` – a double
  - `%%` - the '`%`' character (in printf)

Chapter 3
Relational Operators

Operator	Meaning
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not Equal to

- **Relational expressions** evaluate to the integer values 1 (true) or 0 (false).
- We will use relational operators to form relational expression of conditions.

Three Logical Operators

- Exclamation mark **!**
  - NOT (negation)
  - unary
- Two ampersands **&&**
  - AND (conjunction)
  - binary
- Two vertical pipes **||**
  - OR (inclusive disjunction)
  - Binary

- These operators are used to combine more than one condition forming a complex condition.

The **if** statement: syntax

```
If (expression)
        statement;        //single statement executed
                          //if expression is true

 If (expression)
 {                        //statements inside {} are
                          //executed if expression is true
        statement1;
        statement2;
        ...
        statement n;
 }
```

The **switch** statement: Syntax

```
 switch (expression)
{
      case constant:
                  statement(s);
                  break;
      case constant:
                  statement(s);
                  break;
       /* default is optional*/
      default:
                  statement(s);
}
```

Chapter 4
The while Loop

- The while statement is a general repetition statement with the following syntax:

```
while(expression)
{
    statement(s);
}
```

- The process used to execute this construct is as follows:
  1. Test the expression.
  2. If expression is not 0 (not false)

     execute the statements in the brackets and go back to step 1.

     else

     exit the while statement

The do-while Loop

- The “do… while” loop looks as follows:

```
do {
        statement(s);
} while (condition);
```

- for, and while are called top nested loops this means that the condition is evaluated first than the body of the loop is executed (possibly zero times)
- do … while is called bottom nested, i.e. the condition is evaluated after the loop. This means that the Loop will execute at least once.

The for Loop

- The “for” statement performs the same task as a while statement.
- The “for” statement has its built in counter that must be initialized, tested, and updated as part of the “for” structure.
- It typically looks as follows:

```
int counter;
for (counter = initialValue; counter <= bound; counter = counter + increment)
{
    Statemenet(s);
}
```

  - Where counter must be defined before the for statement.
  - Initial value is starting value, bound is the upper bound to be reached and increment is the amount used to increment the counter.
  - “for” statement may be used in both escalating loops or de-escalating loops.
  - “for” statement is customarily used when the number of repetitions is known but should not be the case.

Chapter 5
Library functions

- Lots of useful functions already written.
- Learn how to use them (arguments, libraries, etc.)

- Input Output:
  - printf(), scanf()

- Character handling:
  - isdigit(), islower(), isupper(), tolower(), toupper()

- Math Functions:
  - sqrt(): square root
  - pow(): exponentiation
  - abs(): absolute value
  - exp(): exponential

Function Prototypes

- Functions must be declared prior to their use
- This is called ***prototyping***
- Prototypes should be placed outside of main function and before the main function starts
- Function prototype tells C:
  - The data type returned by the function
  - The number of parameters received
  - The data types of the parameters
  - The order of the parameters

int addTwoNumbers(int, int);

void printBalance(int);

int createRandomNumber(void);

Function Definitions

- Function definitions implement the function prototypes

```
#include <stdio.h>

int addTwoNumbers(int,int); // function prototype
main() {
   printf("Nothing happening in here");
}
int addTwoNumbers(int operand1, int operand2) {
   return operand1 + operand2;
}                                        //function definition
```

Function Call

```
#include <stdio.h>

int addTwoNumbers(int,int); // function prototype
main() {
   int iResult;
   iResult = addTwoNumbers(5,6);
}
int addTwoNumbers(int operand1, int operand2) {
   return operand1 + operand2;
}                                    //function definition
```

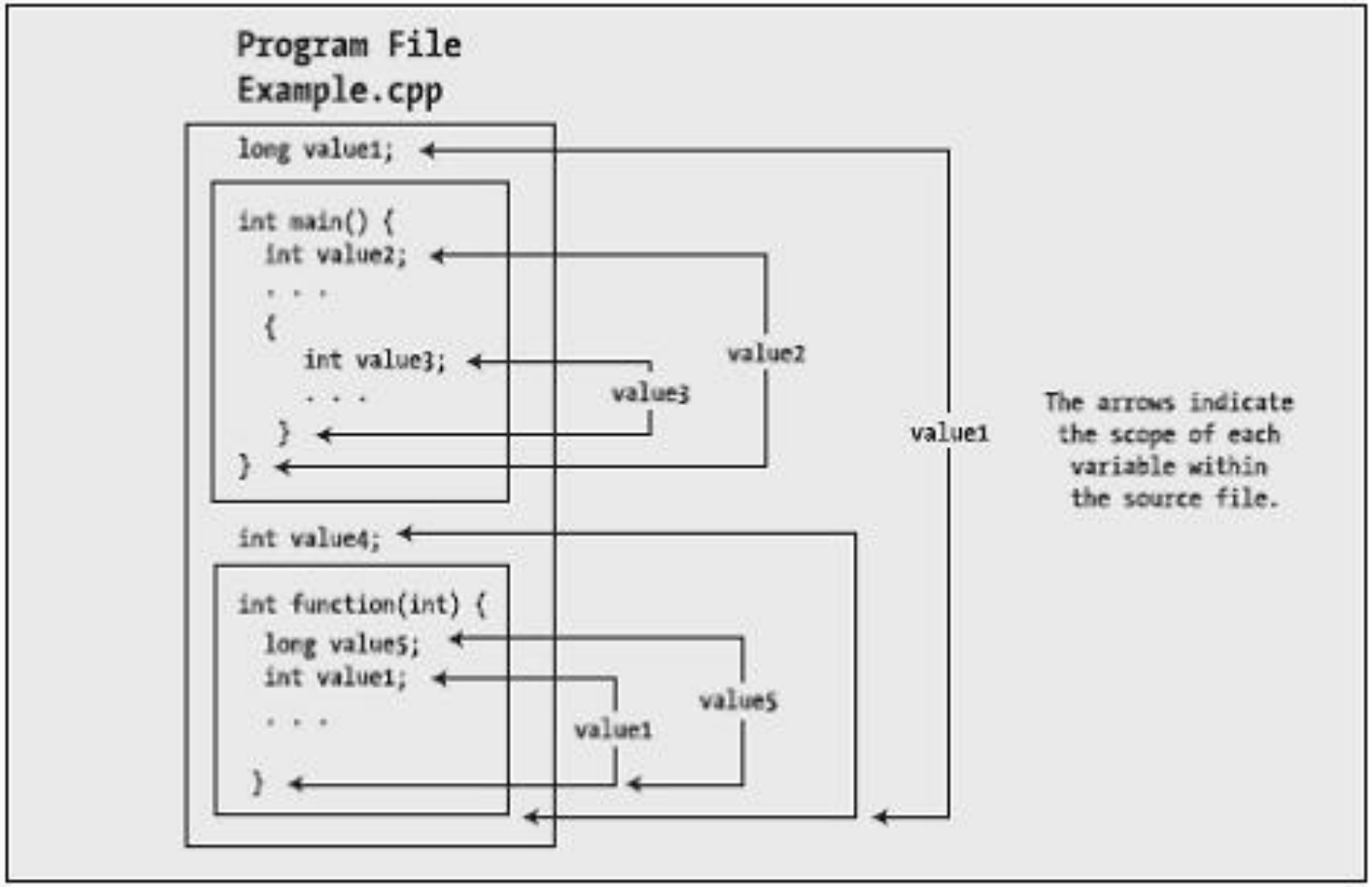
Passing Arguments

- There are two modes for passing information from one function to another: ***by value*** and ***by reference***

- When we pass by **value**:
  - A copy of the data is made in the local variable
  - We pass data
  - The actual variable and the local variable refer to two different addresses
  - There is no change to the actual variable

- When we pass by **reference**:
  - No copy of the data is made
  - We pass the address of the data
  - The actual variable and the local variable refer to the same address
  - A change to the local variable results in the same change to the actual variable

Scope

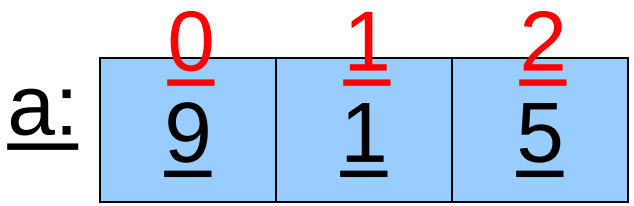

Program File
Example.cpp
long value1;
int main() {
int value2;
. . .
{
int value3;
. . .
}
}
int value4;
int function(int) {
long value5;
int value1;
. . .
}
value2
value3
value1
value5
value1
The arrows indicate the scope of each variable within the source file.

Scope: Local vs. Global

- Based on the scope of variables, they are classified as:
  - **Local Variables:** These are the variables declared within a function, and thus are only accessible within that function. They are local to the function.
  - **Global Variable:** These are the variables defined outside all the functions of a program which makes them accessible to all functions.

Chapter 6
Arrays

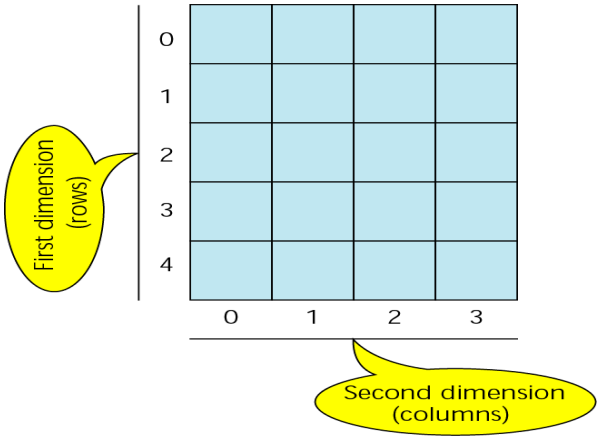
- How do we access each item in the variable; i.e. how do we reference any specific compartment?



- Each cell has a number known as index
- Index specifies the compartment of the variable

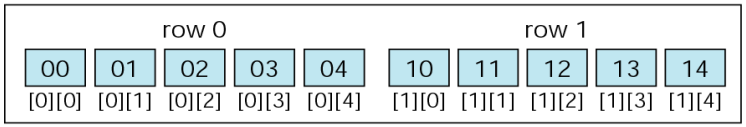
- The proper name of the compartment is the *array cell*

2D Arrays

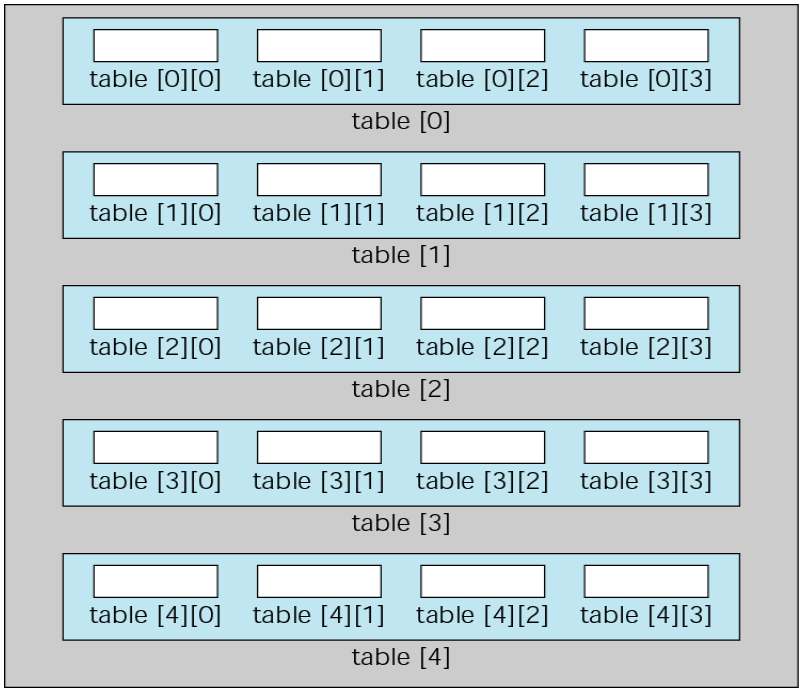


00	01	02	03	04
10	11	12	13	14

User's view



Memory view



table

```
int array[4][3] =
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

(a)

Equivalent

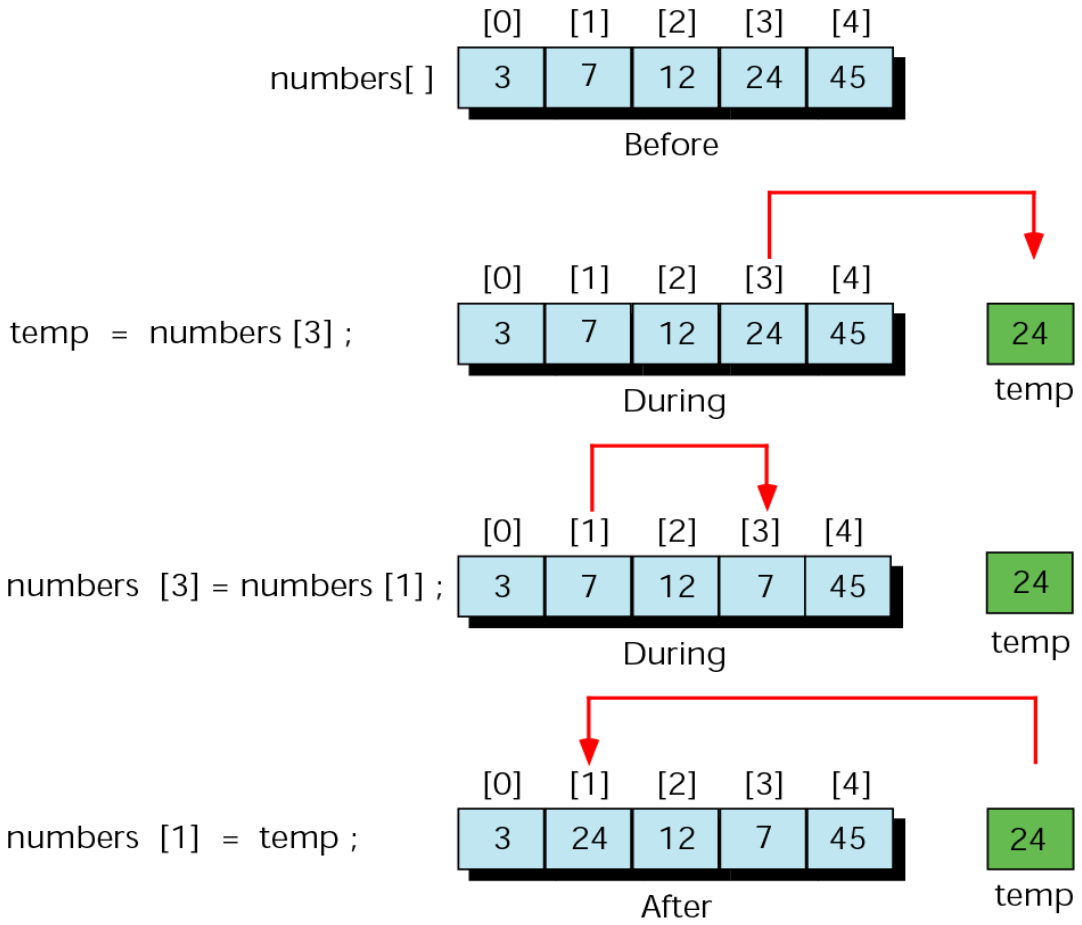
```
int array[4][3];
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;
array[2][0] = 7; array[2][1] = 8; array[2][2] = 9;
array[3][0] = 10; array[3][1] = 11; array[3][2] = 12;
```

(b)

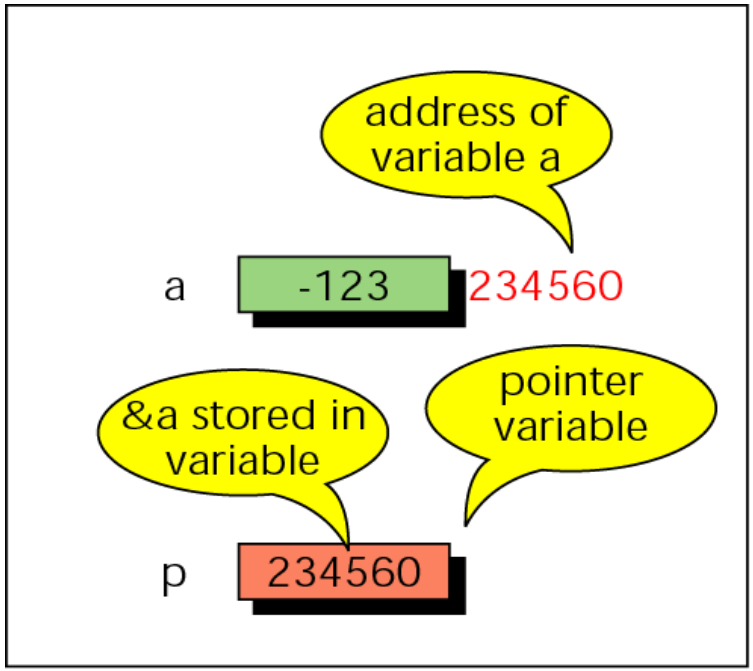
Searching Arrays

- Searching arrays involves using loops
  - Counting loops are used, with counter variable used as the array *index*
  - Each iteration of the loop increments or decrements the *index*. This causes each cell to be examined individually
  - Loop is terminated by one of the two conditions:
    - Every element in the array has been examined
      - Value of *index* has exceeded the array bound
    - The cell that contains the value that satisfies the specified criteria has been found

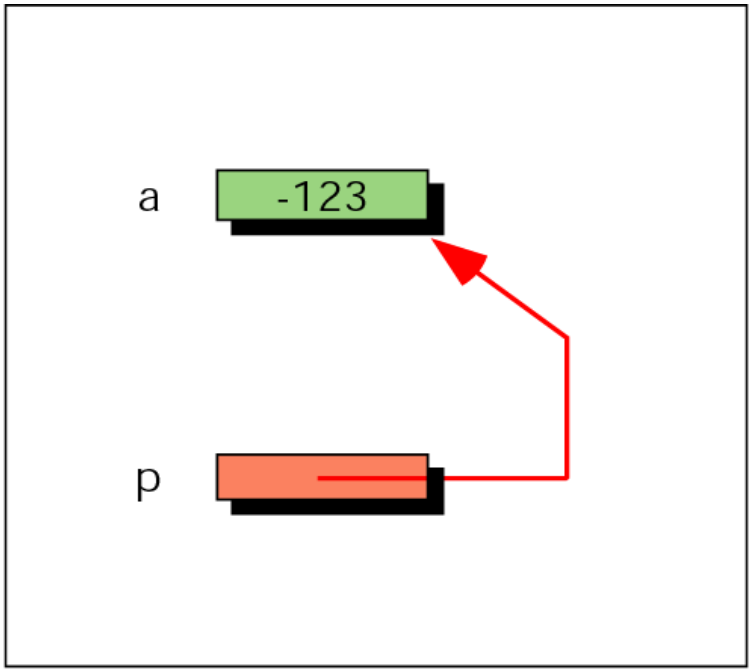
Exchanging Values: the Right Way



Pointer Variable

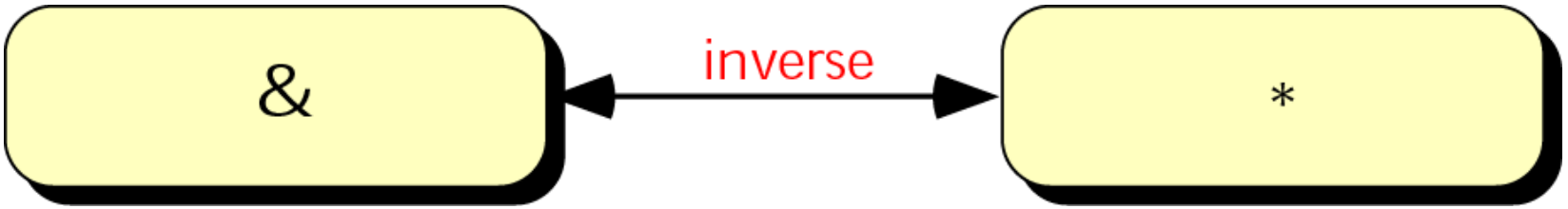


Physical representation

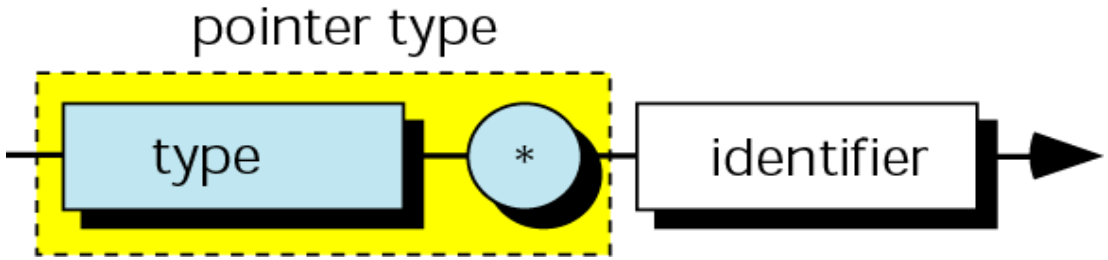


Logical representation

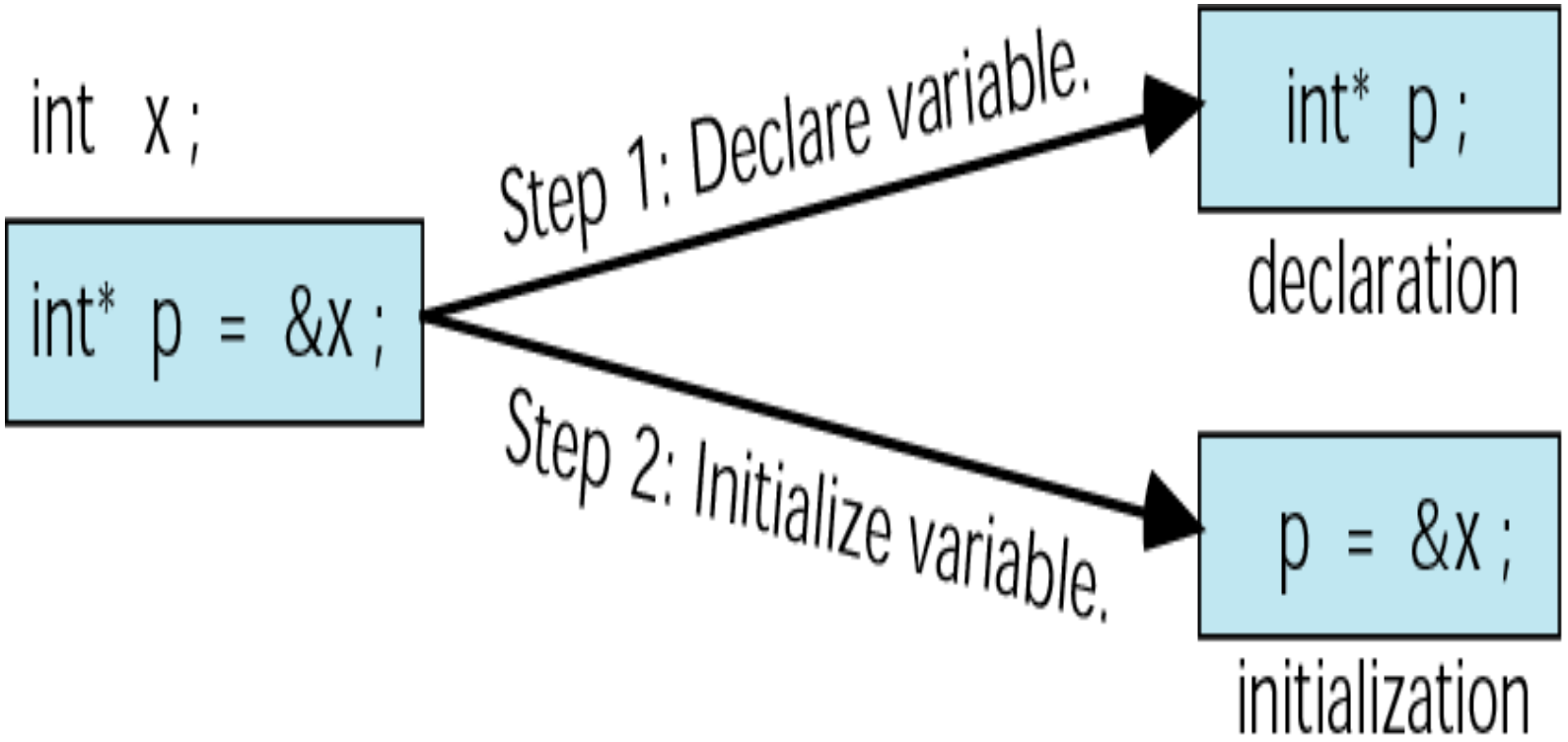
Address and indirection operators



Pointer variable declaration



Initializing Pointer Variables

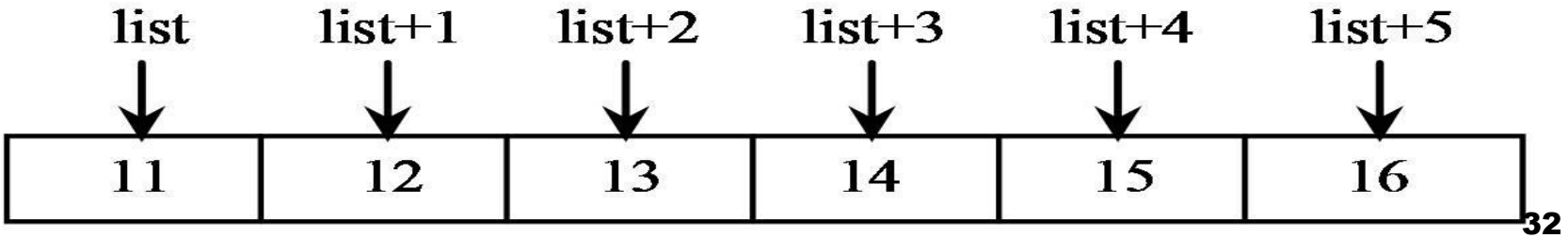


Arrays and Pointers

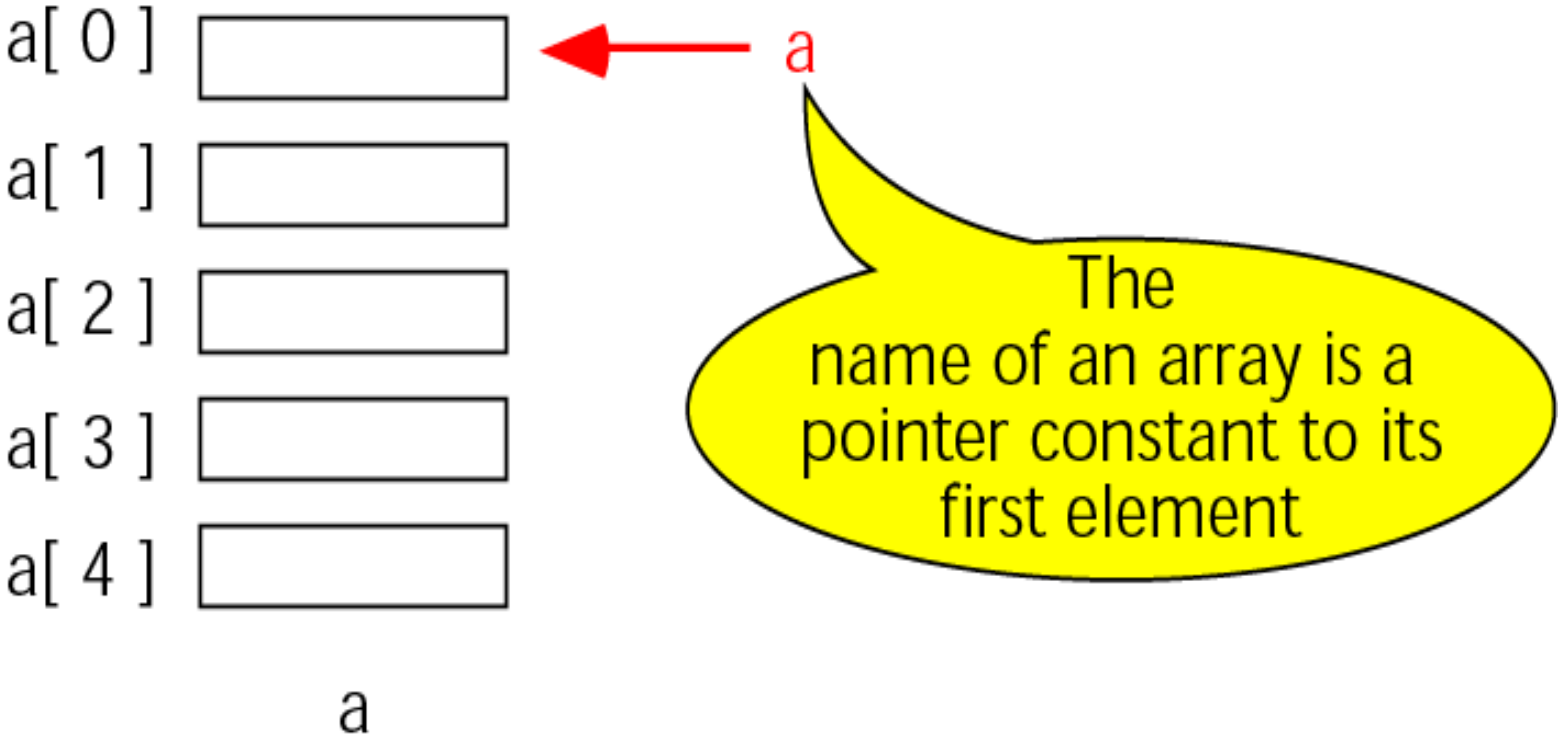
An array variable without a bracket and a subscript actually represents the starting address of the array.

An array variable is essentially a pointer.
Suppose you declare an array of int value as follows:
int list[6] = {11, 12, 13, 14, 15, 16};

*(list + 1) is different from *list + 1. The dereference operator (*) has precedence over +.
So, *list + 1 adds 1 to the value of the first element in the array, while *(list + 1) dereference the element at address (list + 1) in the array.



Pointers to Arrays



Chapter 8
C Strings (Character Arrays)

- There is a difference between `'A'` and `"A"`
  - `'A'` is the character `A`
  - `"A"` is the string `A`
- Because strings are null terminated, `"A"` represents two characters, `'A'` and `'\0'`
- Similarly, `"Hello"` contains six characters, `'H'`, `'e'`, `'l'`, `'l'`, `'o'`, and `'\0'`

Storing and Accessing Strings

- A pointer-based string in C is an array of characters ending in the null terminator ('\0'), which indicates where a string terminates in memory. An array can be accessed via a pointer.
- A string can also be accessed via a pointer, which points to the first character in the string.
- You can declare a string variable using an array or a pointer. For example, the following two declarations are both fine:

```
char city[7] = "Dallas";        // Option 1
char *pCity = "Dallas";         // Option 2
```

Useful Functions

Function	Effect
`strcpy(s1, s2)`	Copies the string `s2` into the string variable `s1` The length of `s1` should be at least as large as `s2`
`strcmp(s1, s2)`	Returns a value < `0` if `s1` is less than `s2` Returns `0` if `s1` and `s2` are the same Returns a value > `0` if `s1` is greater than `s2`
`strlen(s)`	Returns the length of the string `s`, excluding the null character

Chapter 9
Struct Syntax

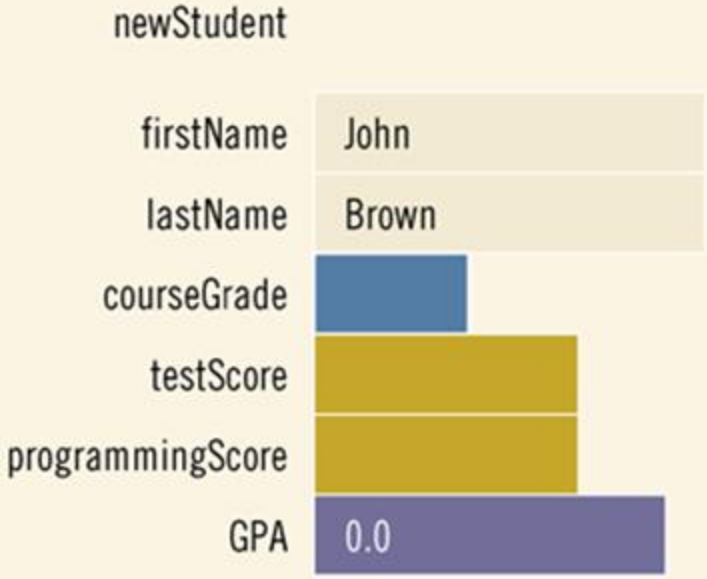
- The general syntax of a `struct` is:

```
struct structName
{
     dataType1 identifier1;
     dataType2 identifier2;
        .
        .
        .
     dataTypen identifiern;
};
```

Accessing `struct` Members

```
newStudent.GPA = 0.0;

newStudent.firstName = "John";
newStudent.lastName = "Brown";
```



User Defined Data Types (typedef)

- The C language provides a facility called *typedef* for creating synonyms for previously defined data type names.
- For example, the declaration:

```
typedef  int  Length;
```

makes the name *Length* a synonym (or alias) for the data type *int.*

- The data "type" name *Length* can now be used in declarations in exactly the same way that the data type *int* can be used:

```
Length a, b, len ;
Length numbers[10] ;
```

Unions

- Like structures, but every member occupies the same region of memory!
  - Structures: members are “working” together
  - Unions: members are “working” one at a time

```
union VALUE {
  float f;
  int i;
  char *s;
};
/* either a float or an int or a string */
```

Type Casting

- Forces one variable of a certain type to be another type
- Might be useful to force integer division
- Example:

```
int x = 12;
int y = 5;
float result = 0;
result = (float) x / (float) y;
```

- Output: 2.4, without type casting 2

Test Format

- True or False
- Multiple Choice
- Trace Code
- Find Errors
- Write Code

True or False

- **struct** is a reserved key word in C

T **F**

Multiple Choice

- Which of the following is NOT a basic C data type?

A) int

B) char

C) string

D) float

Trace Code

- What is the output produced by the following code?

```
#include <stdio.h>

main() {

    printf("Hello World!");

}
```

Find Errors

- Find all errors in the following code segment:

```
#inlude <stdio>

main() [

    Printf('Hello World!');

}
```

Write Code

- Write a function to compute a product of two integers:

Prototype:

Definition:

Call:

The End!