C Pointers, Part 1

Deitel 8th Edition, Chapter 7

Topics, Part 1

Memory Concepts – Review from Week 2

Pointers

Accessing Data in Variables with and without Pointers

Review of Functions

Memory Usage – Arguments to Functions

Memory Usage – Return Values From Functions

Using Pointers – Functions with Output Parameters

Functions with Output Parameters – Class Demo

MEMORY CONCEPTS – REVIEW FROM WEEK 2

DATA IS STORED MEMORY

While a program is executing, data used in the program is stored in memory cells

Remember that memory is **volatile**, so this data is lost when program stops

To Access Memory Cells, We Use Variables

A **variable** is associated with a **memory cell**

Every variable has four components:

1. **Name** – also called an identifier
2. **Data type** – int, char, double, etc.
3. **Value** – the data currently stored in that variable
4. **Location** of the memory cell referred to by this variable name

The Purpose of Each of these Components

Name

- Allows the human writing or reading the code to easily identify a memory cell

Data type

- Tells the operating system how big of a memory cell to use and what kind of values can be stored there

Value

- The data stored in that memory cell

Location

- Where this memory cell is located

Different Data Types Require Different Sized Memory Cells

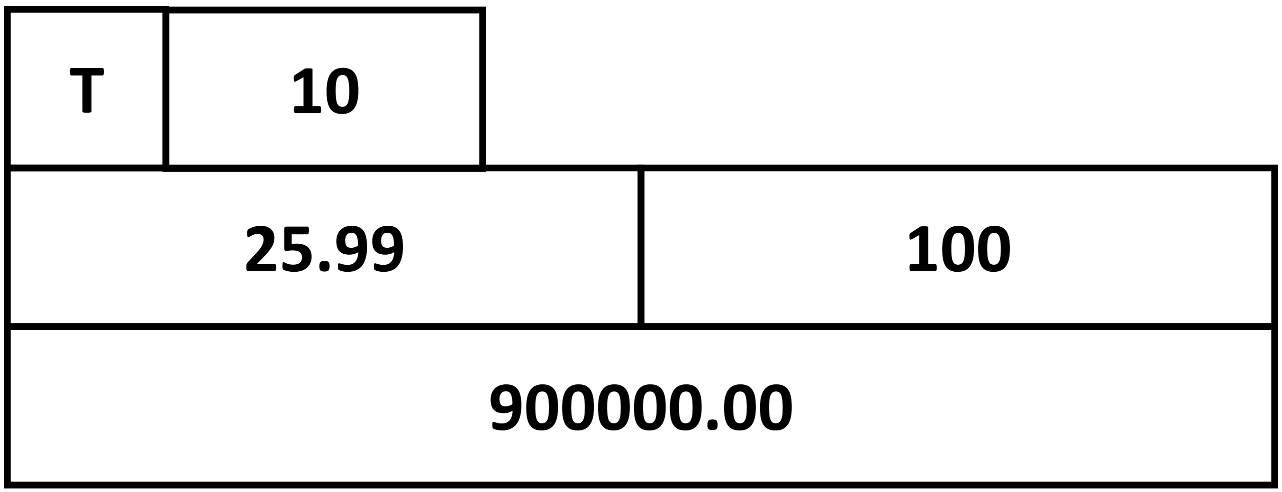
Data Type	Size in bytes
char	1
short integer	2
integer	4
long integer	4
unsigned integer	4
float	4
double	8

Variables and Memory Cell Size Examples

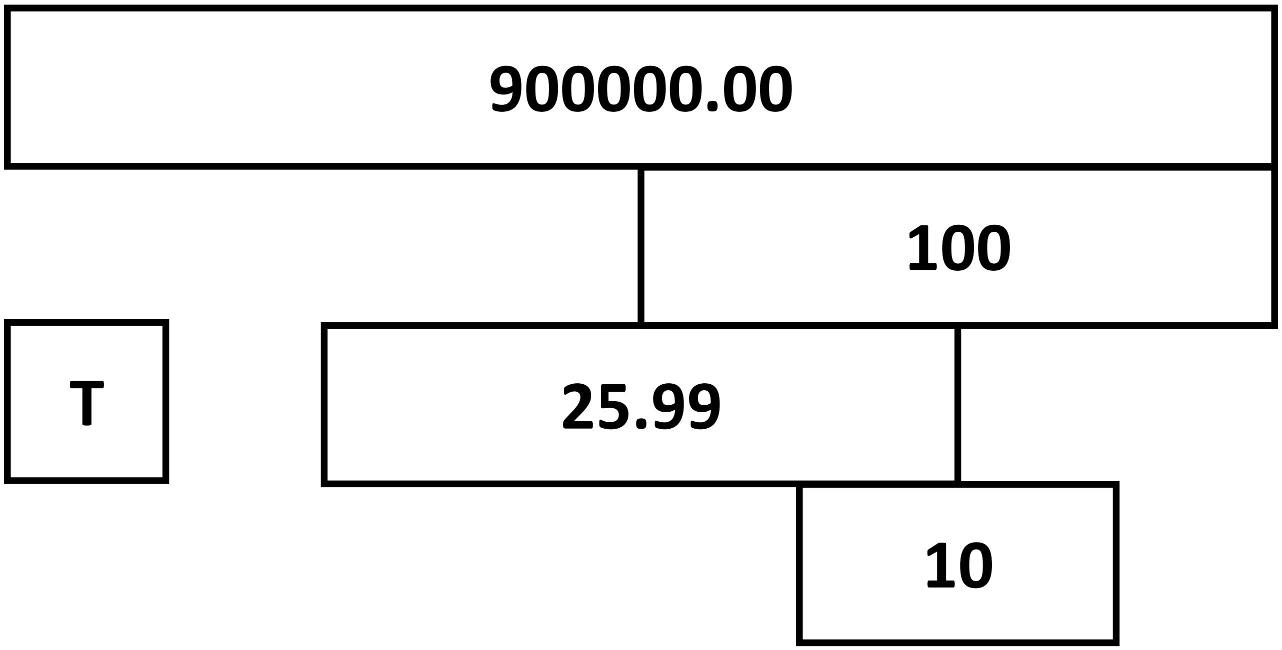
char letter = 'T';	**T**
short num = 10;	**10**
int count = 100;	**100**
float price = 25.99;	**25.99**
double salary = 900000.00	**900000.00**

Remember that these values would be stored in binary format, not plain text.

THESE MEMORY CELLS CAN BE ARRANGED IN VARIOUS WAYS – 1

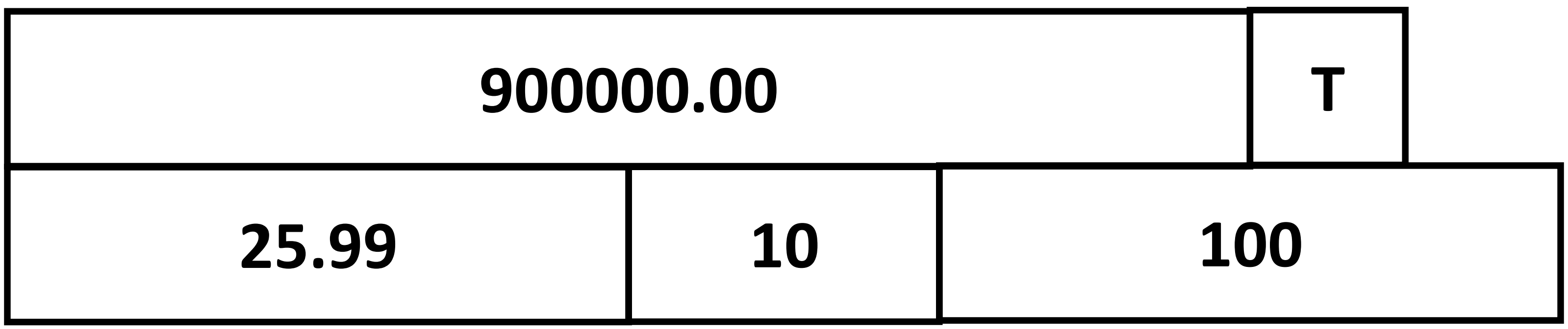


THESE MEMORY CELLS CAN BE ARRANGED IN VARIOUS WAYS – 2



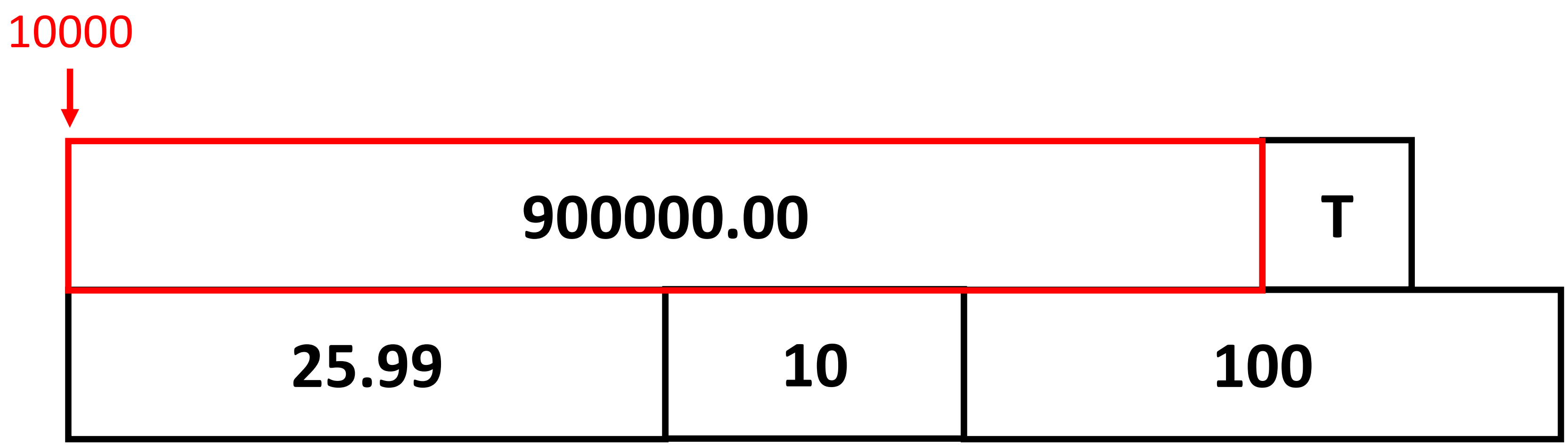
EACH CELL IS LOCATED AT A SPECIFIC LOCATION IN MEMORY

This location is called its **address**.



EACH CELL HAS ITS OWN ADDRESS

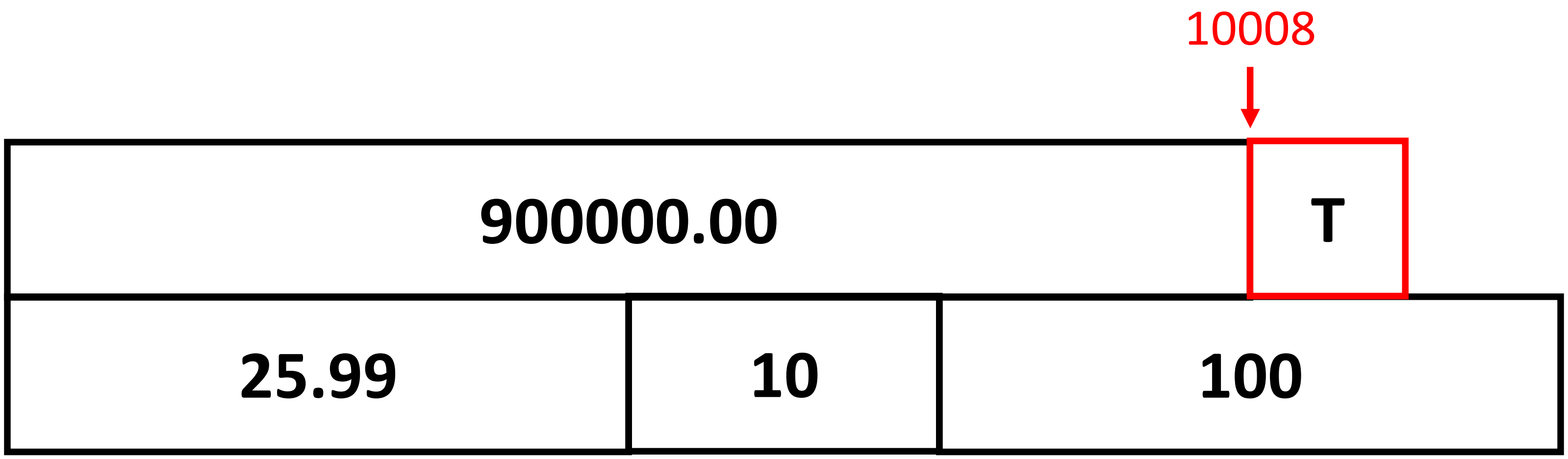
This cell might start at address 10000



Since this cell is 8 bytes long, the next cell's address starts 8 bytes over.

THE NEXT ASSIGNED CELL STARTS 8 BYTES OVER

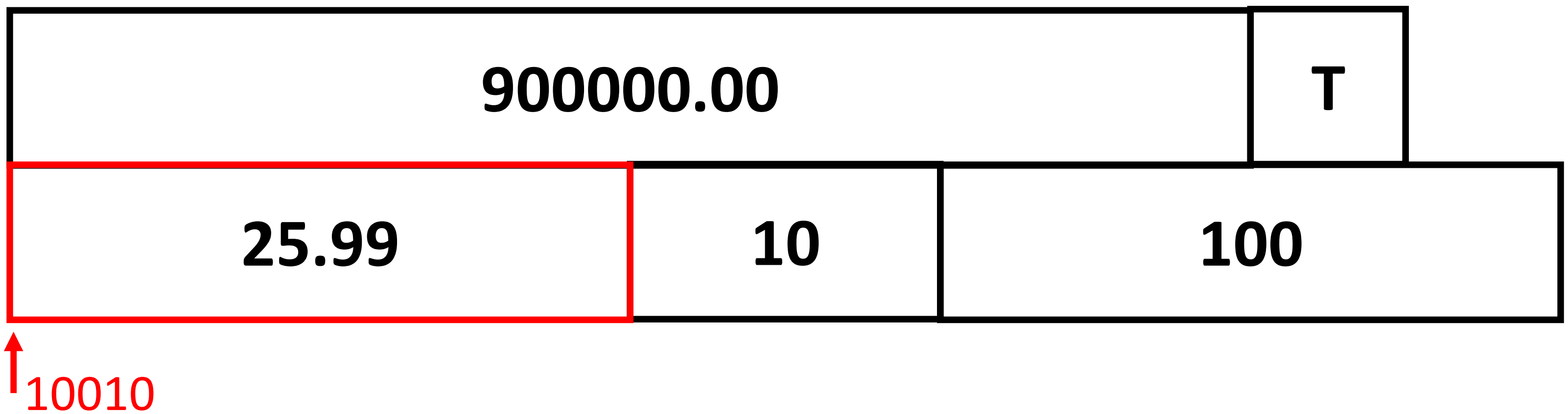
So this cell is at address 10008.



Since this cell is 1 byte long, and there's an empty space next to it that's 1 byte long, the next cell's address starts 2 bytes over.

The Next Assigned Cell Starts 2 Bytes Over

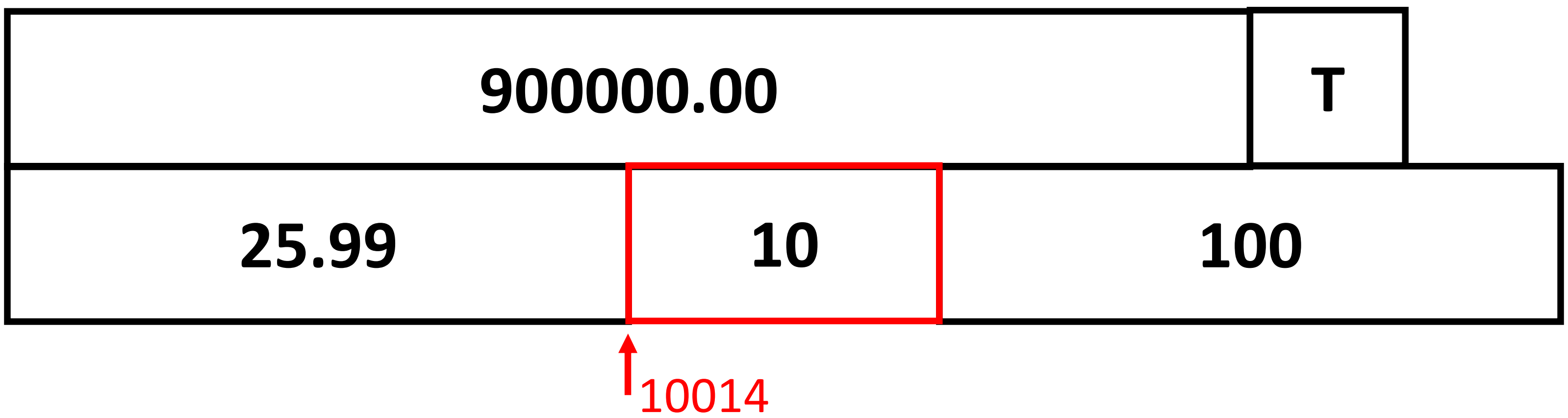
So this cell is at address 10010.



Since this cell is 4 bytes long, the next cell's address starts 4 bytes over.

The Next Assigned Cell Starts 4 Bytes Over

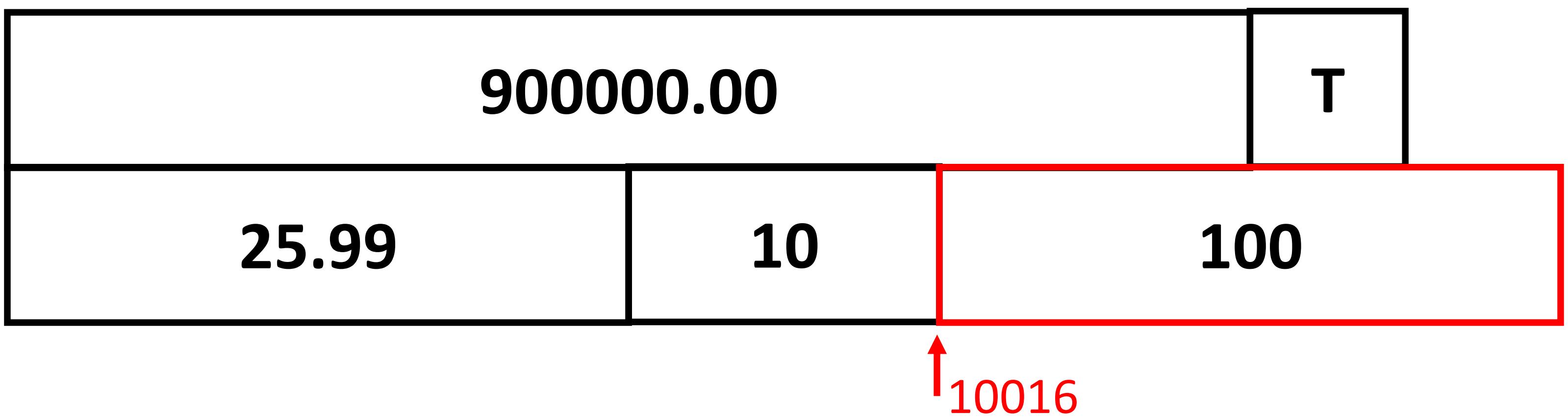
So this cell is at address 10014.



Since this cell is 2 bytes long, the next cell's address starts 2 bytes over.

This Next Assigned Cell Starts 2 Bytes Over

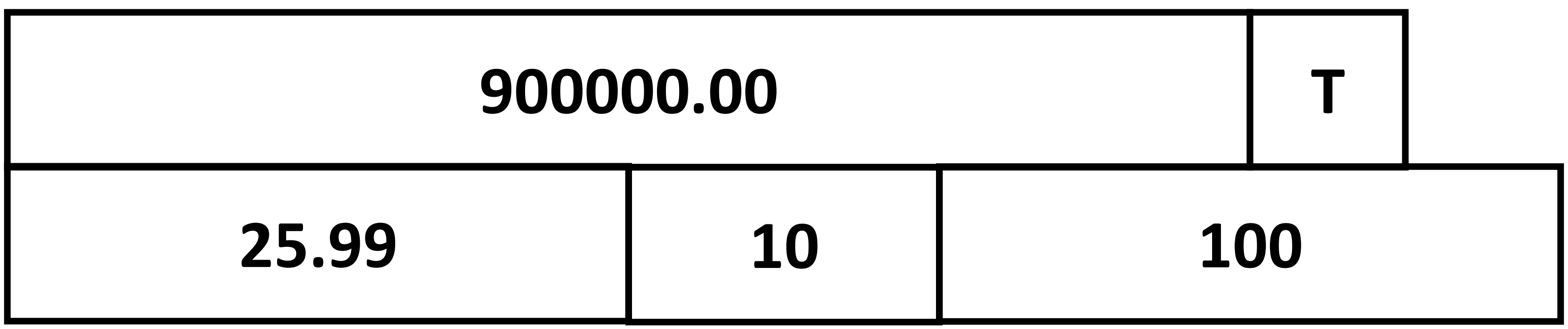
So this cell is at address 10016.



How Are These Cells Located and Accessed by a Program?

The operating system handles it.

When a program uses a variable name, the OS determines which location in memory is matched up with that name.



THE OPERATING SYSTEM KEEPS TRACK OF THESE VARIABLE NAMES & ADDRESSES

In the program, we declare a variable:	**At runtime, OS reserves a cell of the right size at some address**	**And keeps track of the variable name used for the cell**
char letter;	1 byte starting at address 10008	**letter** is at address **10008**
short num;	2 bytes starting at address 10014	**num** is at address **10014**
int count;	4 bytes starting at address 10016	**count** is at address **10016**
float price;	4 bytes starting at address 10010	**price** is at address **10010**
double salary;	8 bytes starting at address 10000	**salary** is at address **10000**

We Can Access These Variables in Two Ways

Using the variable name

- This is how we've been accessing variables so far

Using the variable's address…

POINTERS

To Access a Variable's Address, Use the AddressOf Operator &

Used at the beginning of a variable name, it creates an expression with a value

The value of the expression is the address of the memory cell used by that variable

Example (using count variable from earlier):

The integer variable **count** is at address **10016**

So the expression **&count** has the value 10016

What is a Pointer?

Pointers are **variables that can store addresses of data cells** in use by the program

countPointer (pointer variable)	**100** count (integer variable stored at address 10016)

If we want to save the address of count, we can use a pointer variable

POINTER VARIABLES MUST BE DECLARED TOO

We can't just take any variable and store an address there

A **pointer variable must be declared before use**, just like anything else in a C program

To declare a pointer variable, we must first know the **data type of the cell it will point to**

Only pointers to ints can hold addresses of ints, and only pointers to doubles can hold addresses of doubles, etc.

DECLARING A POINTER VARIABLE

Asterisk in a variable declaration indicates variable is a pointer

Declaration	Meaning	Data Type of Pointer Variable
double * p;	"**p** is a pointer to a double"	pointer to double; or **double ***
char * c;	"**c** is a pointer to a char"	pointer to char; or **char ***
int * j;	"**j** is a pointer to an int"	pointer to int; or **int ***

DECLARING AND ASSIGNING A VALUE TO A POINTER VARIABLE

```
int * countPointer;         //declares the pointer variable
countPointer = &count;      //assigns the address of count
```

10016	**100**
countPointer (pointer variable)	count (integer variable stored at address 10016)

Declaring and Assigning a Value to a Pointer Variable 2

But since we don't really care what the actual address is, we can think of the value of countPointer as just **&count**

&count	**100**
countPointer (pointer variable)	count (integer variable stored at address 10016)

And we say "countPointer is a pointer to count"

Another Example

int x = 25;

int * z;

z = &x;

What is the data type of x?

What is the value of x?

What is the data type of z?

What is the value of z?

Another Example – Results

```
int x = 25;

int * z;

z = &x;
```

What is the data type of x?	integer
What is the value of x?	25
What is the data type of z?	pointer to integer
What is the value of z?	The address of x, or &x

Accessing Data in Variables with and without Pointers

Remember – We Can Access The Data in a Variable in Two Ways

Using the variable name

- This is how we've been accessing variables so far

Using the variable's address

- Via a pointer variable

Direct Access – Using the Variable Name to Access a Value

We can use a variable’s name to refer to a memory cell’s value without having to know the address of the cell

This will copy the value of count1 into a variable called count2:

```
int count2 = count1;
```

This type of access is called **direct access**, because we’re directly accessing memory cells, via their names count1 and count2

Indirect Access – Using the Variable's Address to Access a Value

But we can also refer to a memory cell's value by using its address, if we have a pointer to that memory cell

This will also copy the value of count1 into a variable called count2:

```
int * count_ptr = &count1;

count2 = *count_ptr;
```

This type of access is called **indirect access**, because we're indirectly accessing a value in a cell, via its address

INDIRECTION – DEFINITION

Accessing the contents of a memory cell through a pointer variable that stores its address

In this context, the asterisk is called the **indirection operator**:

```
count2 = *count_ptr;
```

It means “read the value at this address”, or “follow the pointer”

We also say “count_ptr” is **dereferenced**

THREE MEANINGS OF THE ASTERISK *

In this statement:	The asterisk means:	Examples
Variable declaration, prototype, or argument list	The **data type** "pointer to …"	**int *count;** **void functionA (int *c)**
Executable C statement (not arithmetic)	Unary **indirection operator** or "follow the pointer"	**count2 = *pointer;** **(*c)++;**
Executable C statement (arithmetic)	**Multiplication**	**pay = hours * rate;**

Another Indirection Example 1

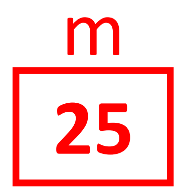
What is happening in memory as these statements execute?

```
int m = 25;

int sum = 0;

int *mPtr;

mPtr = &m;

sum = *mPtr + 5;
```

Another Indirection Example 2

What is happening in memory as these statements execute?

int m = 25;



int sum = 0;

int *mPtr;

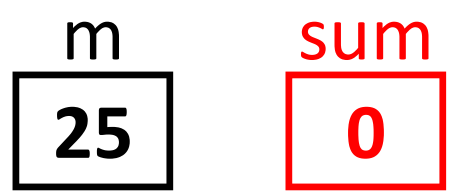
mPtr = &m;

sum = *mPtr + 5;

ANOTHER INDIRECTION EXAMPLE 3

What is happening in memory as these statements execute?

```
int m = 25;

int sum = 0;

int *mPtr;

mPtr = &m;

sum = *mPtr + 5;
```



ANOTHER INDIRECTION EXAMPLE 4

What is happening in memory as these statements execute?

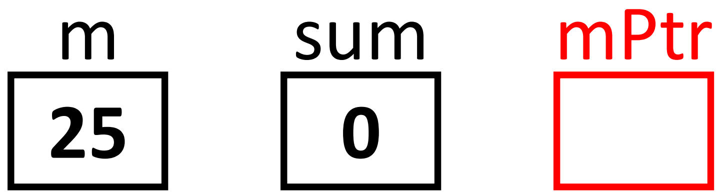
int m = 25;

int sum = 0;

int *mPtr;

mPtr = &m;

sum = *mPtr + 5;



Another Indirection Example 5

What is happening in memory as these statements execute?

```
int m = 25;

int sum = 0;

int *mPtr;

mPtr = &m;

sum = *mPtr + 5;
```

m	sum	mPtr
25	0	&m

Another Indirection Example 6

What is happening in memory as these statements execute?

int m = 25;

int sum = 0;

int *mPtr;

mPtr = &m;

sum = *mPtr + 5;

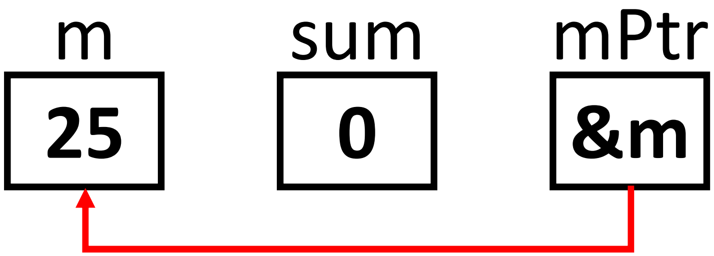
m	sum	mPtr
25	**0**	**&m**

There are four separate actions happening in this statement.

ANOTHER INDIRECTION EXAMPLE 7

What is happening in memory as these statements execute?

```
int m = 25;

int sum = 0;

int *mPtr;

mPtr = &m;

sum = *mPtr + 5;
```



1. **Indirection operator * is associated with variable mPtr**
2. The value in the cell that mPtr is pointing to is read and stored in a temporary location.
3. 5 is added to that value.
4. The result is copied to the cell named sum.

ANOTHER INDIRECTION EXAMPLE 8

What is happening in memory as these statements execute?

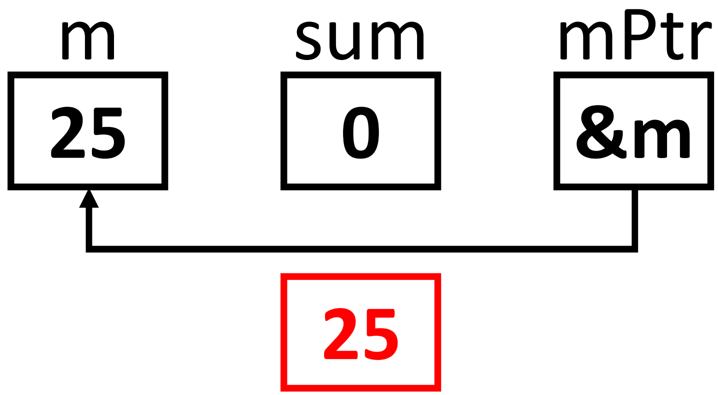
int m = 25;

int sum = 0;

int *mPtr;

mPtr = &m;

sum = *mPtr + 5;

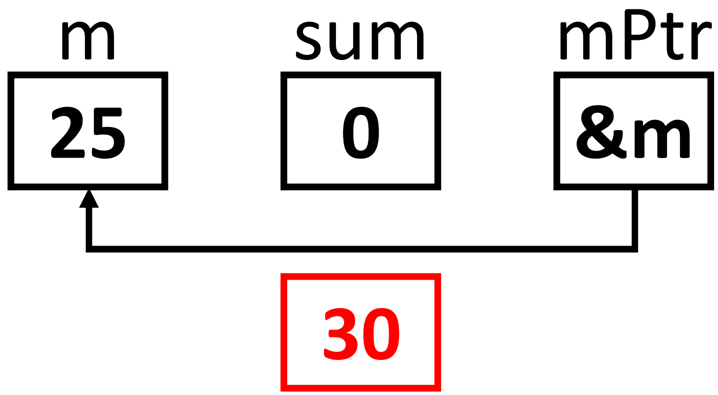


1. Indirection operator * is associated with variable mPtr.
2. **The value in the cell that mPtr is pointing to is read and stored in a temporary location.**
3. 5 is added to that value.
4. The result is copied to the cell named sum.

Another Indirection Example 9

What is happening in memory as these statements execute?

```
int m = 25;

int sum = 0;

int *mPtr;

mPtr = &m;

sum = *mPtr + 5;
```

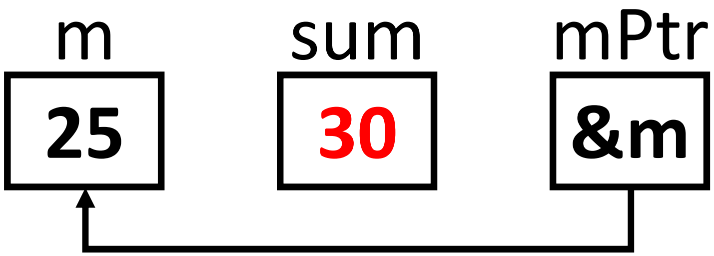


1. Indirection operator * is associated with variable mPtr.
2. The value in the cell that mPtr is pointing to is read and stored in a temporary location.
3. **5 is added to that value.**
4. The result is copied to the cell named sum.

Another Indirection Example 10

What is happening in memory as these statements execute?

```
int m = 25;

int sum = 0;

int *mPtr;

mPtr = &m;

sum = *mPtr + 5;
```



1. Indirection operator * is associated with variable mPtr.
2. The value in the cell that mPtr is pointing to is read and stored in a temporary location.
3. 5 is added to that value.
4. **The result is copied to the cell named sum.**

Indirection = Dereferencing a Pointer

Dereferencing

- The process of accessing the cell pointed to by a pointer variable

Uses the **indirection operator ***

DIRECT VS. INDIRECT – SUMMARY

A variable name **directly** references a value

A pointer **indirectly** references a value

ANOTHER EXAMPLE – WHAT WILL PRINT?

Line 1: **int x = 10;**

Line 2: **int * xPtr = &x;**

Line 3: **printf("%d", *xPtr);**

ANOTHER EXAMPLE – WHAT WILL PRINT – RESULTS

Line 1: **int x = 10;**

Line 2: **int * xPtr = &x;**

Line 3: **printf("%d", *xPtr);**

In line 3, xPtr is **dereferenced** – the value of the cell it points to is accessed

So, 10 would print

TO PRINT AN ADDRESS, USE THE FCS %P

Line 1: **int x = 10;**

Line 2: **int * xPtr = &x;**

Line 3: **printf("%p", xPtr);**

In line 3, xPtr is **not dereferenced** – its own value is read

So, the address of x would print

Demonstrating the & and * Operators

print_pointer.c

Fig07_04.c

& and ***** operators are **complements** of one another

When they're both applied consecutively to aPtr in either order (line 18), the same result is printed

Note the format placeholders in the print_pointer.c example

INITIALIZING A POINTER TO NULL

A pointer may be initialized to **NULL**, **0**, or an **address**

A pointer with the value NULL points to nothing

NULL

- symbolic constant defined in stddef.h and stdio.h

Initializing a pointer to 0 is the same as initializing a pointer to NULL

NULL is preferred – highlights the fact that the variable is a pointer type

nullPointer.c

Review of Functions

Separate Modules in C are Called Functions

We use functions to implement one part of an algorithm

So far, we know that functions can have:

1. **Inputs** – none, one or many (What are these called?)
2. **Outputs** – none, or one value (What is this called?)

SEPARATE MODULES IN C ARE CALLED FUNCTIONS – INPUTS/OUTPUTS

We use functions to implement one part of an algorithm

So far, we know that functions can have:

1. **Inputs** – none, one or many (What are these called?) **parameter(s)**
2. **Outputs** – none, or one value (What is this called?) **return value**

CALLING A FUNCTION WITH INPUT VALUES

When we want a function to begin executing, we **call** it, using a statement called a **function call**

If the function requires input value(s), those values must be provided at this time, or the code will not compile

These values are called **arguments**, and represent actual data

Arguments provided in a function call must match in _______, ______ & _____ to what the function requires in its parameter list

Calling a Function with Input Values – NOT Acronym

When we want a function to begin executing, we **call** it, using a statement called a **function call**

If the function requires input value(s), those values must be provided at this time, or the code will not compile

These values are called **arguments**, and represent actual data

Arguments provided in a function call must match in **number**, **order** & **type** to what the function requires in its parameter list

FUNCTION ARGUMENTS ARE COPIES

When we pass a value to a function via an argument, that value is **COPIED** from the argument in the call into the parameter in the function header

The function works on the **COPIED VALUE, NOT THE ORIGINAL VARIABLE**

argument_example.c

MEMORY USAGE – ARGUMENTS TO FUNCTIONS

MEMORY USAGE – ARGUMENTS TO FUNCTIONS – STEP 1

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10

MEMORY USAGE – ARGUMENTS TO FUNCTIONS – STEP 2

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	

MEMORY USAGE – ARGUMENTS TO FUNCTIONS – STEP 3

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	

MEMORY USAGE – ARGUMENTS TO FUNCTIONS – STEP 4

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = 10

MEMORY USAGE – ARGUMENTS TO FUNCTIONS – STEP 5

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = 10
21	print c	10	incrementCount	Value of c is printed	

MEMORY USAGE – ARGUMENTS TO FUNCTIONS – STEP 6

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = 10
21	print c	10	incrementCount	Value of c is printed	
22	c++		incrementCount	Value of c is incremented	incrementCount: c = 11

Memory Usage – Arguments to Functions – Step 7

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = 10
21	print c	10	incrementCount	Value of c is printed	
22	c++		incrementCount	Value of c is incremented	incrementCount: c = 11
23	print c	11	incrementCount	Value of c is printed	

MEMORY USAGE – ARGUMENTS TO FUNCTIONS – STEP 8

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	Count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = 10
21	print c	10	incrementCount	Value of c is printed	
22	c++		incrementCount	Value of c is incremented	incrementCount: c = 11
23	print c	11	incrementCount	Value of c is printed	
24	}		incrementCount	End of incrementCount function; control transfers back to main	Memory associated with incrementCount is released

MEMORY USAGE – ARGUMENTS TO FUNCTIONS – STEP 9

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
12	print count	10	main	print count	

MEMORY USAGE – ARGUMENTS TO FUNCTIONS – STEP 10

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
12	print count	10	main	print count	
14	return(0)		main	main ends	Memory associated with main is released

Memory Usage – Arguments to Functions – Step 11

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS

The Result of a Function is the Return Value

If we want to use the result of a function's work in the calling code, so far, we can see only one thing – the return value

This is called "returning a value"

argument_example_return.c

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 1

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 2

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 3

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	count = incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 4

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	count = incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = 10

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 5

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	count = incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = 10
21	print c	10	incrementCount	Value of c is printed	

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 6

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	count = incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = 10
21	print c	10	incrementCount	Value of c is printed	
22	c++		incrementCount	Value of c is incremented	incrementCount: c = 11

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 7

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	count = incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = 10
21	print c	10	incrementCount	Value of c is printed	
22	c++		incrementCount	Value of c is incremented	incrementCount: c = 11
23	print c	11	incrementCount	Value of c is printed	

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 8

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	Count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	count = incrementCount(count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = 10
21	print c	10	incrementCount	Value of c is printed	
22	c++		incrementCount	Value of c is incremented	incrementCount: c = 11
23	print c	11	incrementCount	Value of c is printed	
25	return (c);		incrementCount	End of incrementCount function; control transfers back to main; **value of c is returned**	Memory associated with incrementCount is released

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 9

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	**count = incrementCount(count);**		main	**Control returns here to complete the assignment**	main: **count = 11**

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 10

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	count = incrementCount(count);		main		main: count = 11
12	print count	11	main	print count	

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 11

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	count = incrementCount(count);		main		main: count = 11
12	print count	11	main	print count	
14	return(0)		main	main ends	Memory associated with main is released

MEMORY USAGE – RETURN VALUES FROM FUNCTIONS – STEP 12

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)

THE RESULT OF A FUNCTION WITH ONE RESULT

Return values work for a single result

But what if we want to return **more than a single value** to the calling code?

USE POINTERS TO:

Return **more than one result** to the calling code

Allow a function to work on the **original value**, not a copy

Pass large objects to functions, since making copies of a big area of memory can be wasteful

Using Pointers – Functions with Output Parameters

RETURNING MORE THAN ONE RESULT

Remember our earlier question: What if we want to return more than one piece of data from a function?

REMEMBER ARGUMENT LISTS…

Communication between the calling code and a function

So far, our communication using arguments has been one-sided:

calling code → function

If we use **addresses in the argument list**, we can “send back” more than one piece of information via arguments in the list

This is called an **output parameter**

ARGUMENTS PASSED BY VALUE

Remember that a value passed to a function via an argument is a COPY of the original value

But, a copy of an ADDRESS is the SAME ADDRESS as the original.

It leads to the SAME MEMORY CELL.

So the function (using indirect access) acts on the ORIGINAL VALUE, in the ORIGINAL MEMORY CELL, because it has a copy of its address.

FUNCTION WITH OUTPUT PARAMETERS

argument_example_output.c

MEMORY USAGE – OUTPUT PARAMETERS TO FUNCTIONS – STEP 1

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10

MEMORY USAGE – OUTPUT PARAMETERS TO FUNCTIONS – STEP 2

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	

Memory Usage – Output Parameters to Functions – Step 3

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(&count);		main	Function call – main transfers control to incrementCount; main is paused	

MEMORY USAGE – OUTPUT PARAMETERS TO FUNCTIONS – STEP 4

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(&count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int * c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = &count

MEMORY USAGE – OUTPUT PARAMETERS TO FUNCTIONS – STEP 5

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(&count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int * c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = &count
21	print *c	10	incrementCount	*c dereferenced; value at that address is printed	

MEMORY USAGE – OUTPUT PARAMETERS TO FUNCTIONS – STEP 6

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = **11**
10	print count	10	main	print count	
11	incrementCount(&count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int * c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = &count
21	print *c	10	incrementCount	*c dereferenced; value at that address is printed	
22	(*c)++		incrementCount	*c dereferenced; value at address is incremented	incrementCount: c = &count

Memory Usage – Output Parameters to Functions – Step 7

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 11
10	print count	10	main	print count	
11	incrementCount(&count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int * c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = &count
21	print *c	10	incrementCount	*c dereferenced; value at that address is printed	
22	(*c)++		incrementCount	*c dereferenced; value at address is incremented	incrementCount: c = &count
23	print *c	11	incrementCount	*c dereferenced; value at that address is printed	

MEMORY USAGE – OUTPUT PARAMETERS TO FUNCTIONS – STEP 8

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 11
10	print count	10	main	print count	
11	incrementCount(&count);		main	Function call – main transfers control to incrementCount; main is paused	
19	incrementCount(int * c)		incrementCount	Value of argument count is copied to parameter c	incrementCount: c = &count
21	print *c	10	incrementCount	*c dereferenced; value at that address is printed	
22	(*c)++		incrementCount	*c dereferenced; value at address is incremented	incrementCount: c = &count
23	print *c	11	incrementCount	*c dereferenced; value at that address is printed	
24	}		incrementCount	End of incrementCount; control transfers back to main	Memory associated with incrementCount is released

MEMORY USAGE – OUTPUT PARAMETERS TO FUNCTIONS – STEP 9

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 11
10	print count	10	main	print count	
11	incrementCount(&count);		main	Control returns to main; nothing left to do in this statement	
12	print count	11	main		

MEMORY USAGE – OUTPUT PARAMETERS TO FUNCTIONS – STEP 10

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)
8	int count = 10		main	count is assigned 10	main: count = 10
10	print count	10	main	print count	
11	incrementCount(&count);		main		
12	print count	11	main		

MEMORY USAGE – OUTPUT PARAMETERS TO FUNCTIONS – STEP 11

Line #	Statement	Output	Current Function	What Happens	Call stack (memory)

PASS-BY-VALUE VS. PASS-BY-REFERENCE

Another example – two versions of same functionality – to cube an integer

Fig07_06.c

- Uses pass-by-value

Fig07_07.c

- Uses pass-by-reference

POINTER PARAMETERS

Header and prototype for cubeByReference specify **int *** parameter

cubeByReference receives the **address of an integer variable** as an argument, stores the address locally in nPtr and does not return a value

FUNCTIONS WITH OUTPUT PARAMETERS – CLASS DEMO 1

Write a program that computes the sum and the product of two numbers, using a function.

The function receives the two numbers as input parameters, and "sends back" the sum and the product using output parameters.

So the function has four parameters in total:

- Two input parameters
- Two output parameters

FUNCTIONS WITH OUTPUT PARAMETERS – CLASS DEMO 2

Write a program that computes the sum and the product of two numbers, using a function.

The function receives the two numbers as input parameters, and "sends back" the sum and the product using output parameters.

So the function has two parameters in total:

- Two output parameters that also act as input parameters