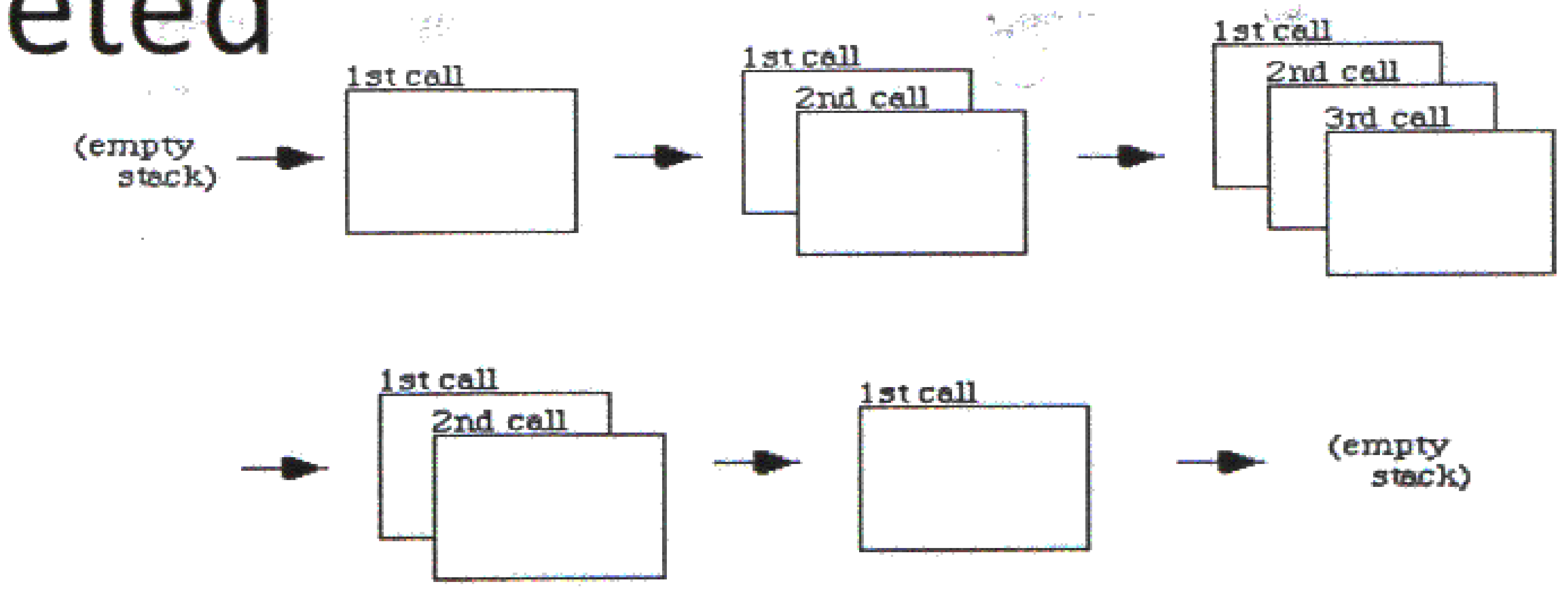
Recursive Function

- C++ arranges the memory spaces needed for each function call in a *stack.*
- The memory area for each new call is placed on the top of the stack
- And then taken off again when the execution of the call is completed



Example

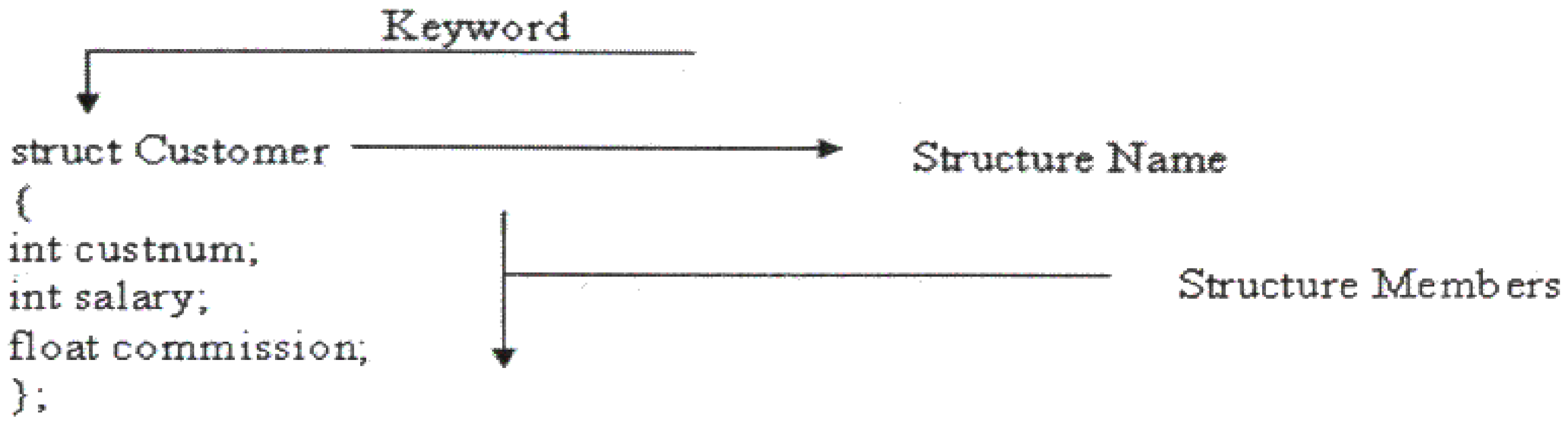
- recursive function sums the first n elements of an integer array "a[]".
- int sum_of(int a[], int n)
- {
- if (n == 1) return a[0];
- else
- return (a[n-1] + sum_of(a,n-1));
- }

Structures

- **What is a Structure?**
- Structure is a collection of variables under a single name.
- Variables can be of any type: int, float, char etc. The main difference between structure and array is that arrays are collections of the same data type and structure is a collection of variables under a single name.

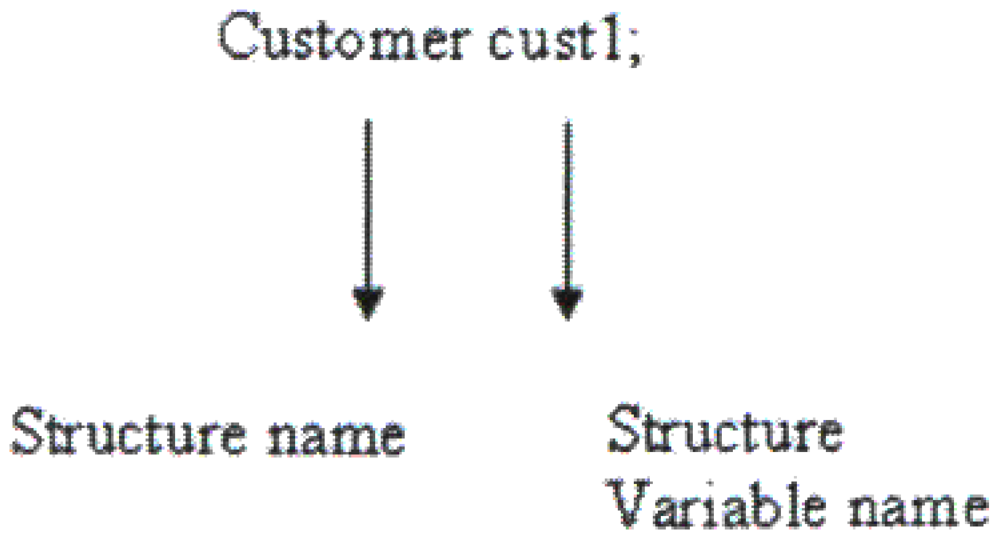
How to declare and create a Structure

- Three variables: *custnum* of type int, *salary* of type int, *commission* of type float are structure members and the structure name is Customer. This structure is declared as follows:



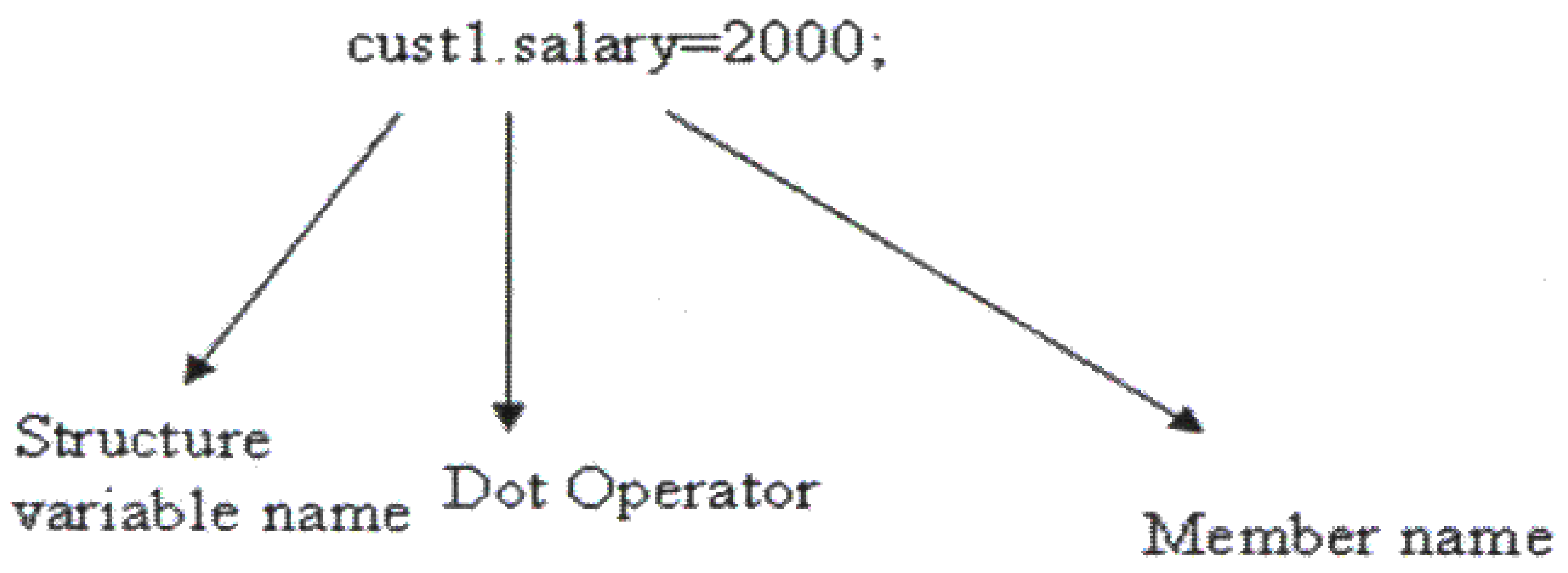
```
struct Customer
{
int custnum;
int salary;
float commission;
};
```

- **How to declare Structure Variable?**



```
Customer cust1;
```

How to access structure members in C++?

- A programmer wants to assign 2000 for the structure member *salary* in the above example of structure *Customer* with structure variable *cust1* this is written as:
- 

```
cust1.salary=2000;
```

Pointers

Outline

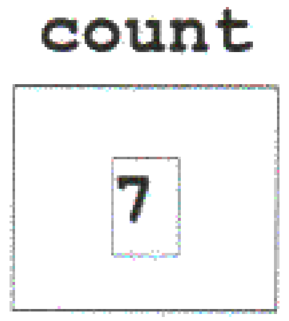
- Introduction
- Pointer Variable Declarations and Initialization
- Pointer Operators
- Calling Functions by Reference
- Using const with Pointers
- Pointer Expressions and Pointer Arithmetic
- Relationship Between Pointers and Arrays

Introduction

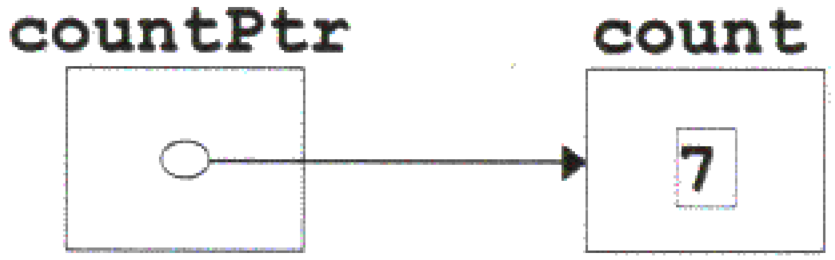
- Pointers
  - Powerful, but difficult to master
  - Simulate pass-by-reference
  - Close relationship with arrays and strings

Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as values
  - Normally, variable contains specific value (direct reference)



  - Pointers contain address of variable that has specific value (indirect reference)



- Indirection
  - Referencing value through pointer
- Pointer declarations
  - `*` indicates variable is pointer

```
int *myPtr;
```

  declares pointer to **`int`**, pointer of type **`int *`**

  - Multiple pointers require multiple asterisks

```
int *myPtr1, *myPtr2;
```

Pointer Variable Declarations and Initialization

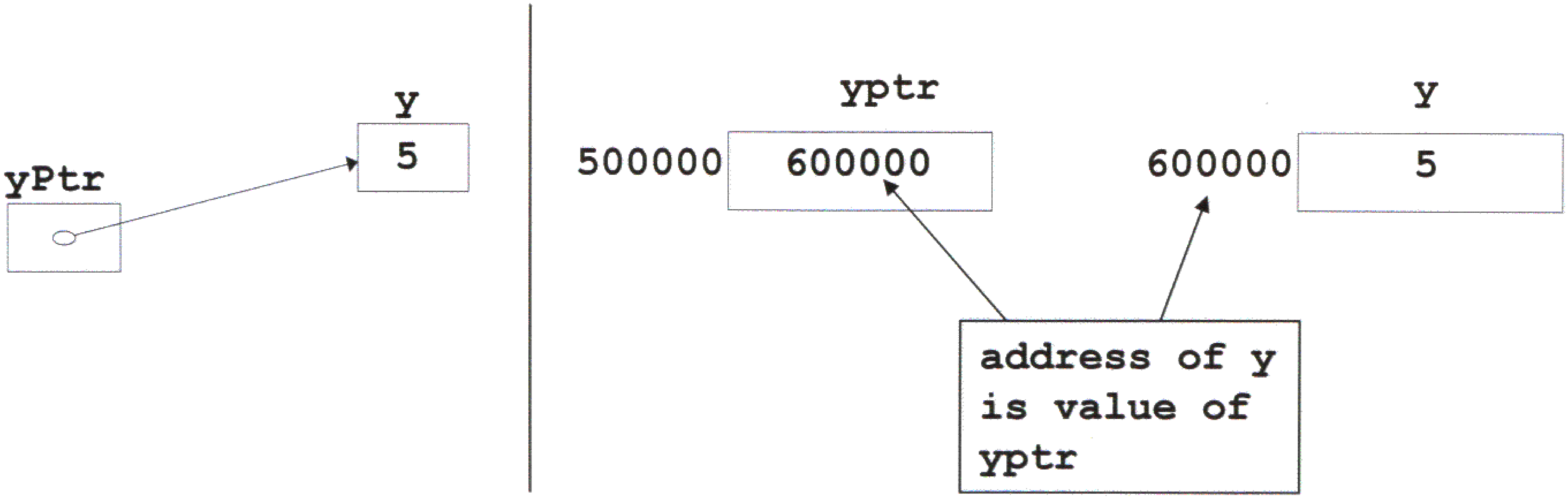
- Can declare pointers to any data type
- Pointer initialization
  - Initialized to **0**, **NULL**, or address
    - **0** or **NULL** points to nothing

Pointer Operators

- **&** (address operator)
  - Returns memory address of its operand
  - Example

```
int y = 5;
int *yPtr;
yPtr = &y;    // yPtr gets address of y
```

  - **yPtr** “points to” **y**



Pointer Operators

- * (indirection/dereferencing operator)
  - Returns synonym for object its pointer operand points to
  - ***yPtr** returns **y** (because **yPtr** points to **y**).
  - dereferenced pointer is lvalue
    - ***yptr = 9; // assigns 9 to y**
- * and **&** are inverses of each other

```
// Fig. 5.4: fig05_04.cp
// Using the & and * operators.
#include <iostream>
 using std::cout;
   using std::endl;

   int main()
   {
     int a;      // a is an integer
     int *aPtr;  // aPtr is a pointer to an integer

     a = 7;
     aPtr = &a;  // aPtr assigned address of a

     cout << "The address of a is " << &a
         << "\nThe value of aPtr is " << aPtr;

     cout << "\n\nThe value of a is " << a
         << "\nThe value of *aPtr is " << *aPtr;

     cout << "\n\nShowing that * and & are inverses of "
         << "each other.\n&*aPtr = " << &*aPtr
         << "\n*&aPtr = " << *&aPtr << endl;

```

* and & are inverses of each other

```
   return 0;  // indicates successful termination

} // end main
```

```
The address of a is 0012FED4
The value of aPtr is 0012FED4

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012FED4
*&aPtr = 0012FED4
```

***** and **&** are inverses; same result when both applied to **`aPtr`**