Ve 280

Programming and Elementary Data Structures

Review of C++ Basics

Learning Objectives:

Freshen your memory of basics C++ (lvalue/rvalue, function declaration vs definition, function call mechanism, array, pointer vs reference, struct…)

Very Basic Concepts

- Variables
- Built-in data types, e.g., `int`, `double`, etc.
- Input and output, e.g., `cin`, `cout`.
- Operators
  - Arithmetic: +, -, *, etc.
  - Comparison: <, >, ==, etc.
  - x++ versus ++x
- Flow of controls
  - Branch: if/else, switch/case
  - Loop: while, for, etc.

An Example

```
#include <iostream>
using namespace std;
int main() {
   // Calculating the area of a square
   int length, area;
   cin >> length;
   if(length > 0) {
      area = length * length;
      cout << "area is " << area << endl;
   }
   else
      cout << "negative length!" << endl;
   return 0;
}
```

lvalue and rvalue

- Two kinds of expressions in C++
  - **lvalue**: An expression which may appear as either the left-hand or right-hand side of an assignment
  - **rvalue**: An expression which may appear on the right- but not left-hand side of an assignment

- E.g., any non-constant variable is an lvalue.
- Any constant is an rvalue.

Which statements are correct?

Select all the correct answers. Variables `a` and `b` are of type `int` and `c` is an array.

- **A.** `10` is an lvalue.
- **B.** `a+1` is an lvalue.
- **C.** `a+b` is an lvalue.
- **D.** `c[2*3]` is an rvalue.
- **E.** None of the above.

Function Declarations vs. Definitions

- Function declaration (or function prototype)
  - Shows how the function is called.
  - Must appear in the code before the function can be called.
  - Syntax:

```
Return_Type Function_Name(Parameter_List);
//Comment describing what function does
```

```
int add(int a, int b); //Comment
```

- Function definition
  - Describes how the function does its task.
  - Can appear before or after the function is called.
  - Syntax:

```
Return_Type  Function_Name(Parameter_List)
{
  //function code
}
```

```
int add(int a, int b) {
      return (a + b);
}
```

Function Declaration

- Tells:
  - return type
  - how many arguments are needed
  - types of the arguments
  - name of the function
  - formal parameter names

Type Signature

Formal Parameter Names

- Example:

```
double total_cost(int number, double price);
// Compute total cost including 5% sales tax on
// number items at cost of price each
```

Function Definition

- Provides the same information as the declaration
- Describes how the function does its task

- Example:

function header

```
double total_cost(int number, double price)
{
    double TAX_RATE = 0.05; //5% tax
    double subtotal;
    subtotal = price * number;
    return (subtotal + subtotal * TAX_RATE);
}
```

function body

Function Call Mechanisms

- Two mechanisms:
  - Call-by-Value
  - Call-by-Reference

```
void f(int x)
{
   x *= 2;
}
```

```
void f(int& x)
{
   x *= 2;
}
```

```
int main()
{
   …
   int a=4;
   f(a);
   …
}
```

What will a be?

Array

- An array is a fixed-sized, indexed data type that stores a collection of items, all of the same type.
- Declaration: `int b[4];`
- Accessing array elements using index: `b[i]`
- C++ arrays can be passed as arguments to a function.

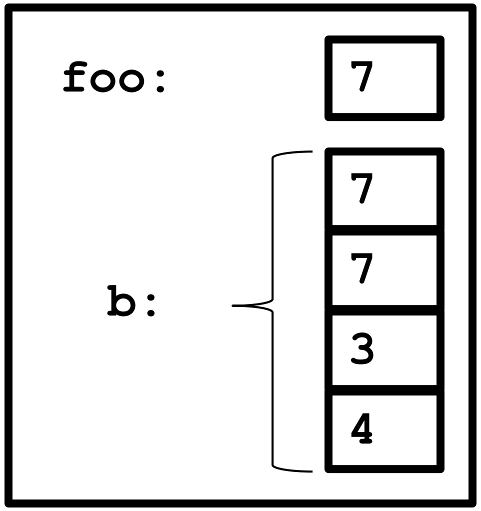
```
int sum(int a[], unsigned int size);
   // Returns the sum of the first
   // size elements of array a[]
```

Array is passed by **reference**.

Array as Function Argument

Using the values below, what would the contents of `b` be after calling `add_one(b, 4)`?

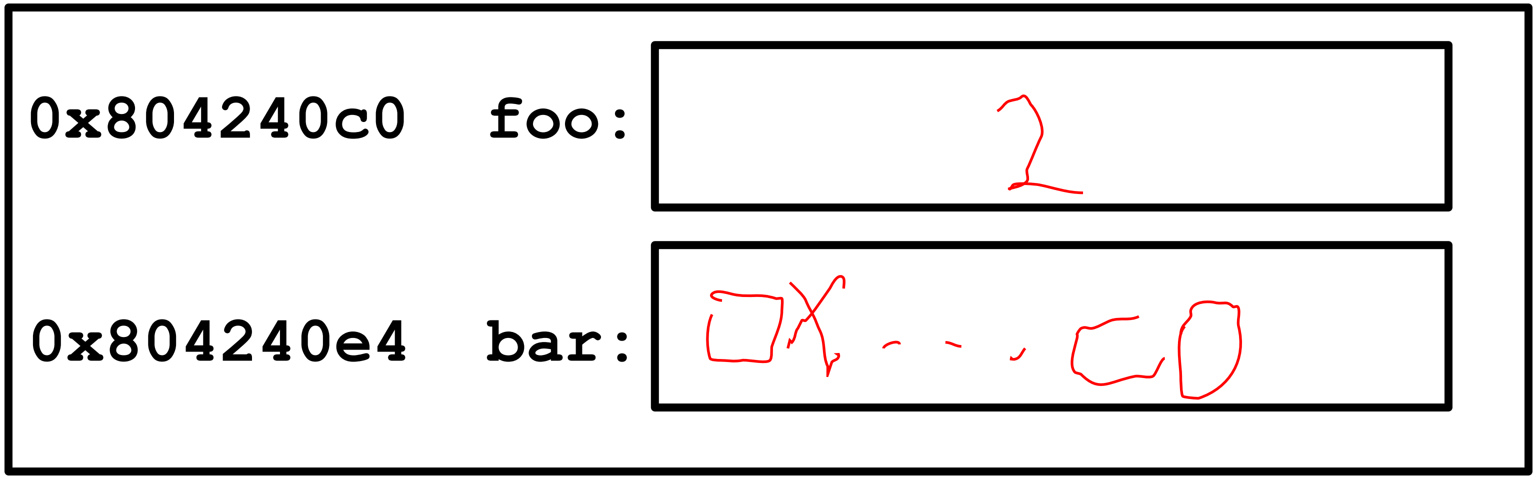
```
void add_one(int a[], unsigned int size) {
   unsigned int i;
   for (i=0; i<size; i++) {
      a[i]++;
   }
}
```



- **A.** 8, 8, 4, 4 **B.** 7, 7, 3, 4
- [✓ **C.** 8, 8, 4, 5] **D.** None of the above.

Pointers: Working with Addresses

```
int foo = 1;
int *bar;    // Define a pointer
bar = &foo; // addressing operation
*bar = 2;    // dereference operation
```



References

- **Reference** is an **alternative** name for an object.

```
int iVal = 1024;
int &refVal = iVal;
```

- refVal is a reference to iVal. We can change iVal through refVal.

- Reference **must be initialized** using a **variable** of the same type.

```
int &refVal2; // Error: not initialized
int &refVal3 = 10; // Error: 10 is not
                   // a variable
```

References

- There is **no way to rebind** a reference to a different object after initialization.

```
int iVal = 1024;
int &refVal = iVal;
int iVal2 = 10;
refVal = iVal2;
```

- refVal still binds to iVal, not iVal2.

Pointers Versus References

- Both pointers and references allow you to pass objects by reference.

- Any differences between pointers and references?
  - Pointers require some extra syntax at calling time (&), in the argument list (*), and with each use (*); references only require extra syntax in the argument list (&).
  - You can change the object to which a pointer points, but you cannot change the object to which a reference refers.
    - In this sense, pointer is **more flexible**

What are the final values?

Select all the correct answers. A and C correspond to the left example, while B and D to the right one.

```
int x = 0;
int &r = x;
int y = 1;
r = y;
r = 2;
```

```
int x = 0;
int *p = &x;
int y = 1;
p = &y;
*p = 2;
```

- [✓ **A.** x = 2, y = 1, r = 2] **B.** x = 0, y = 1, *p = 2
- **C.** x = 0, y = 1, r = 2 **D.** x = 2, y = 2, *p = 2

Pointers

Why use them?

- You might wonder why you'd ever want to use pointers, since they require extra typing, and is error-prone.

- There are (at least) two reasons to use pointers:

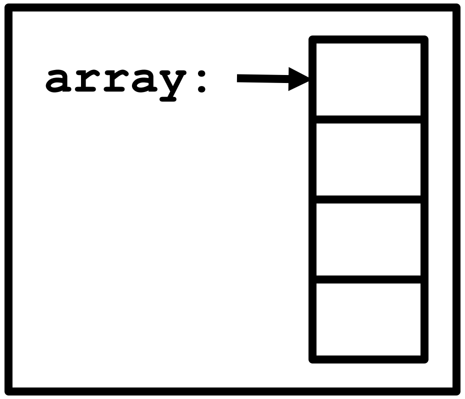
  1. They provide a convenient mechanism to work with arrays.

  2. They allow us to create structures (unlike arrays) whose size is not known in advance.

Pointers and Arrays

- If you look at the **value** of the variable `array` (not `array[0]`) you would find that it would be exactly the same as the **address** of `array[0]`.
- In other words,

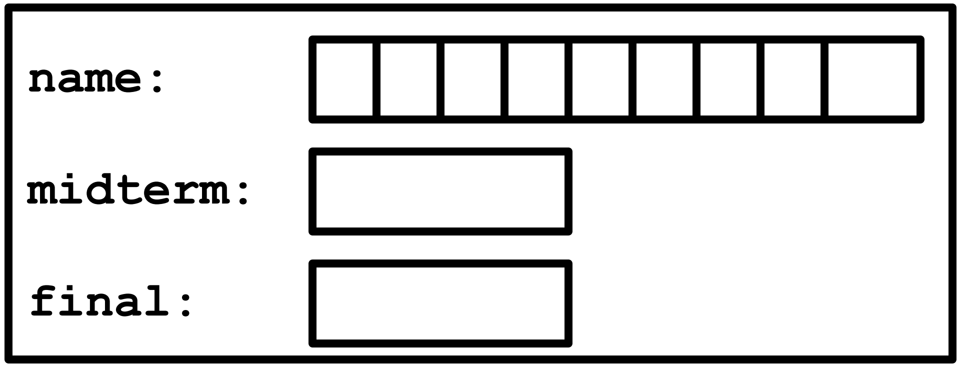
```
array == &array[0]
```



Structs

- Declare a `struct` type that holds grades.
- Why struct? To create a **compound type**

```
struct Grades {
   char name[9];
   int  midterm;
   int  final;
};
```

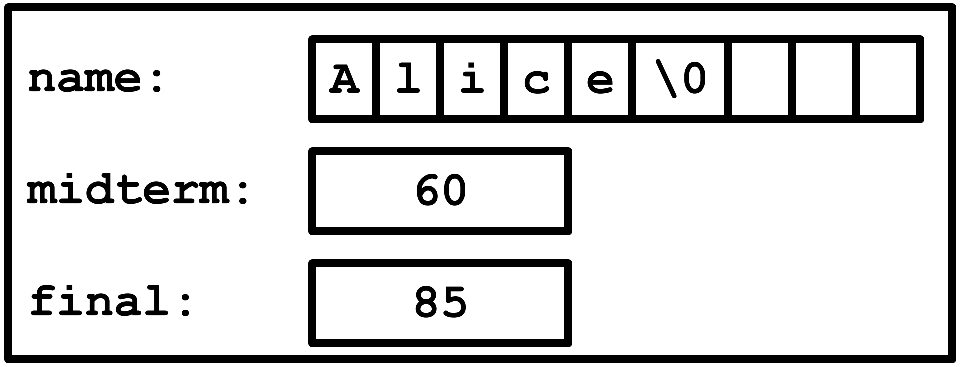


- This statement declares the **type** "struct grades", but does not declare any **objects** of that type.
- We can define single objects of this type as follows:

```
struct Grades alice;
```

Structs

```
struct Grades {
   char name[9];
   int  midterm;
   int  final;
};
```

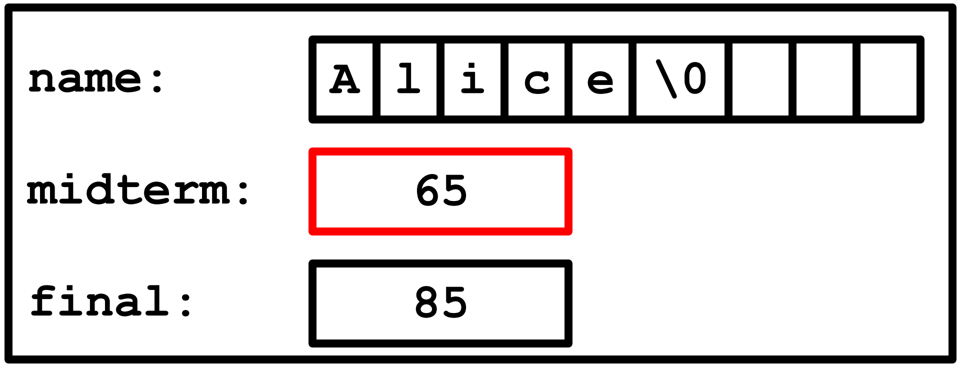


- We can initialize them in the following way:

```
struct Grades alice= {"Alice", 60, 85};
```

Structs

```
struct Grades {
   char name[9];
   int  midterm;
   int  final;
};
```



- Once we have a struct, we can access its individual components using the "dot" operator:

  `alice.midterm = 65;`

  - This changes the `midterm` element of `alice` to 65

- If you have a pointer to struct, visit component using "`->`"

```
struct Grades *gPtr = &alice;
gPtr->final = 90;
```

Reference

- Pointers
  - Problem Solving with C++, 8th Edition, Chapter 9.1
- References
  - C++ Primer, 4th Edition, Chapter 2.9