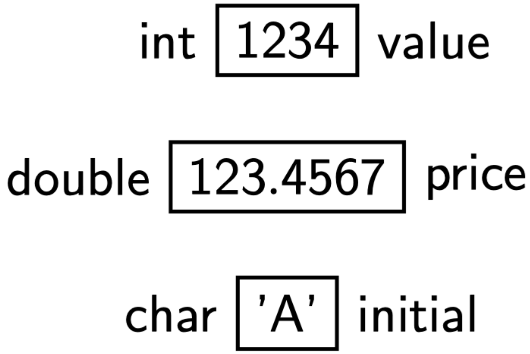
Foundation of Computer Science: Pointers

Dr Kafi Rahman, PhD
Assistant Professor @Computer Science
Truman State University

Local Variable: Recap

- A variable is allocated exactly enough memory to hold one value of the declared type



int 1234 value

double 123.4567 price

char 'A' initial

```
int value;
double price;
char initial;
```

Variables in Memory

- a computer's memory is a list of numbered locations, each of which refers to a byte of 8 bits
    - the number of a byte is its address
- a simple variable (e.g., int or double) refers to a portion of memory containing a number of consecutive bytes
    - the number of bytes is determined by the type of the variable (e.g., on ice, 4 bytes for unsigned, 8 bytes for double)
- the address of the variable is the address of the first byte where it is located

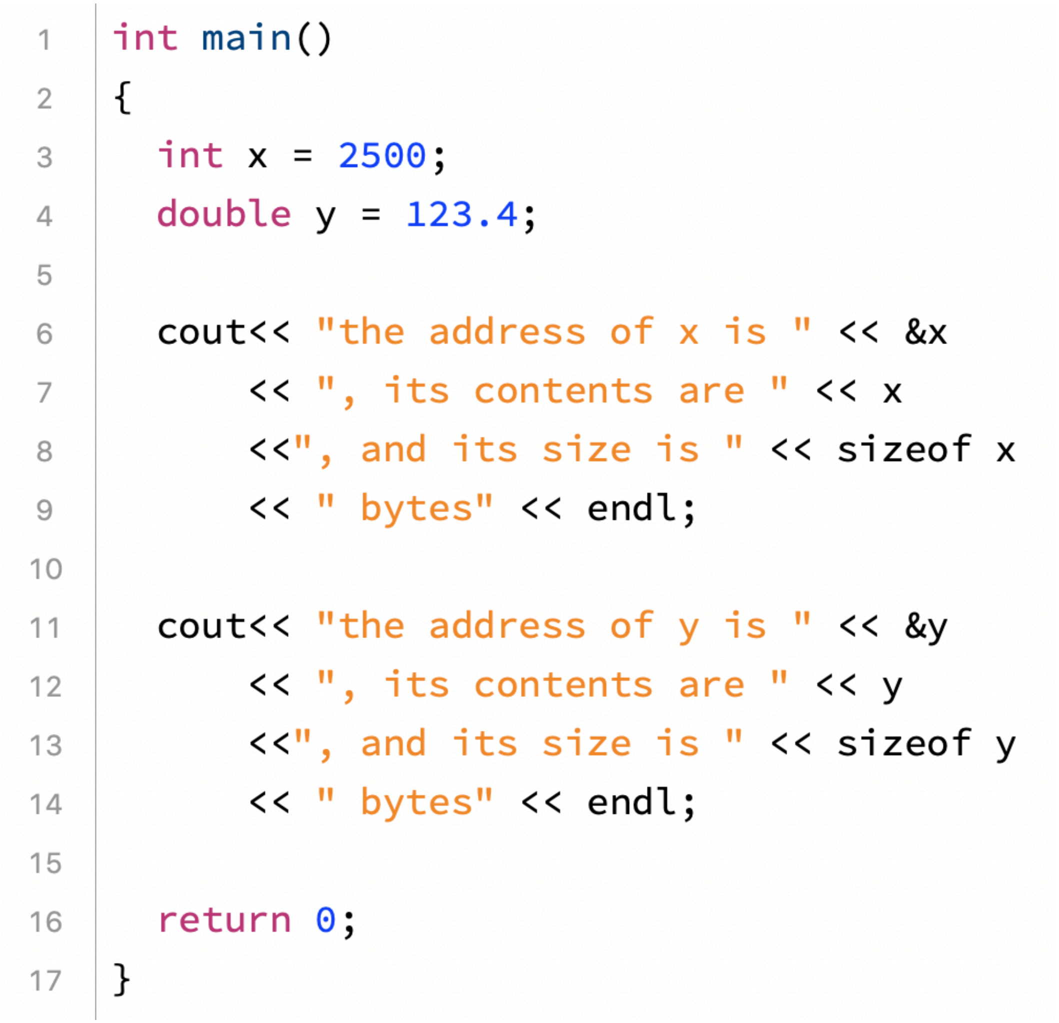
Address Operator

- when you use a variable in a program, the compiler assumes you want the contents of that variable's location in memory
- but sometimes you actually want the address of the variable in memory
- sometimes you also want to know how many bytes of memory a variable occupies
- there is a way to do each of these (not surprisingly)

Address Operator (cont)

- to get a variable's address, we use the address-of operator: &
- to get the number of bytes a variable holds, we use the sizeof operator (it looks like a function, but it is really an operator)

Address Operator (cont)

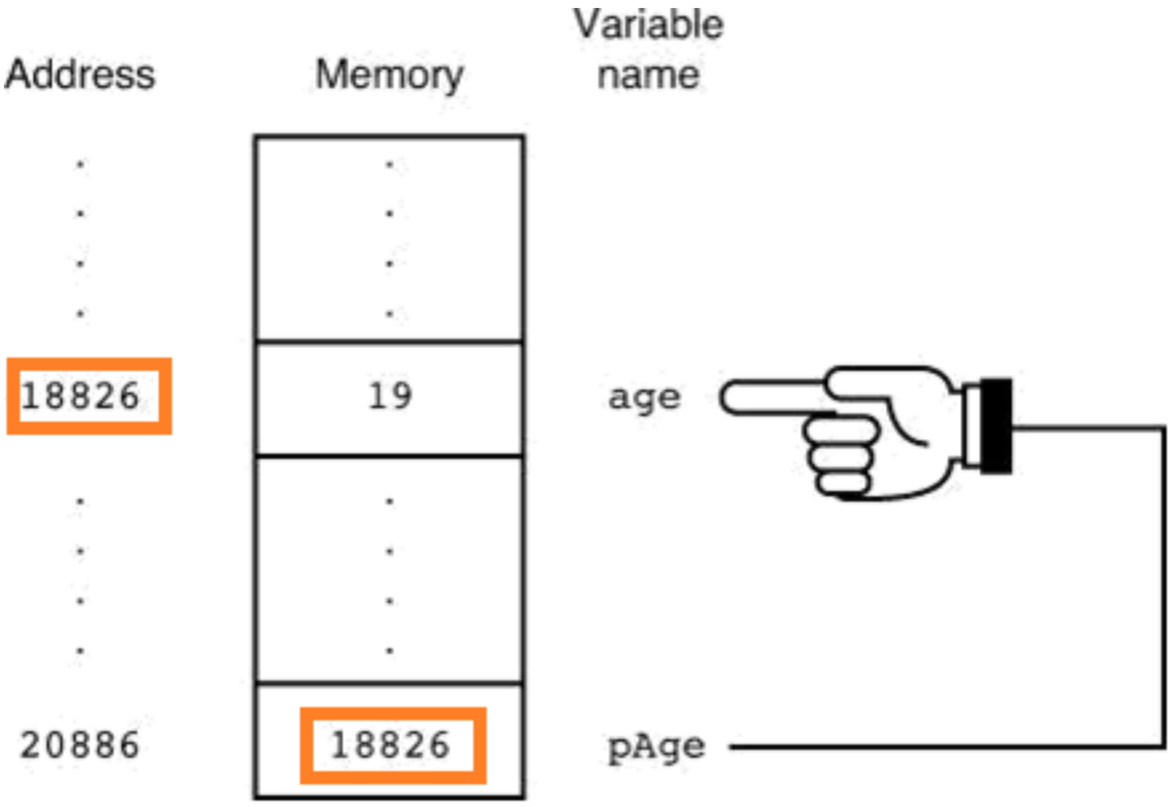


```
int main()
{
  int x = 2500;
  double y = 123.4;

  cout<< "the address of x is " << &x
      << ", its contents are " << x
      <<", and its size is " << sizeof x
      << " bytes" << endl;

  cout<< "the address of y is " << &y
      << ", its contents are " << y
      <<", and its size is " << sizeof y
      << " bytes" << endl;

  return 0;
}
```

Pointer Variable

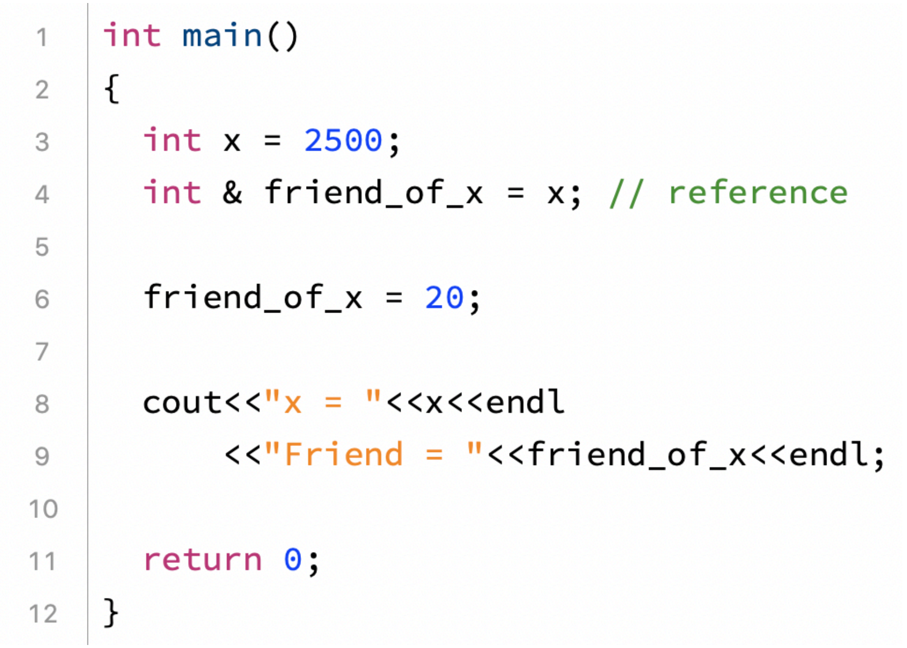
- a pointer variable aka pointer is a variable that holds a memory address
  - just as the purpose of an int is to hold an integer
  - and a double is to hold a double
  - the purpose of a pointer is to hold an address
- this allows you to indirectly reference a memory location though the use of a variable that "points to" another location

Pointer Variable



Reference Variable

- you have used variables that refer to other locations already a reference parameter is an alias for the "real" variable that is located in the calling scope
- here friend_of_x is a reference variable. Any update to this variable will update the original variable.

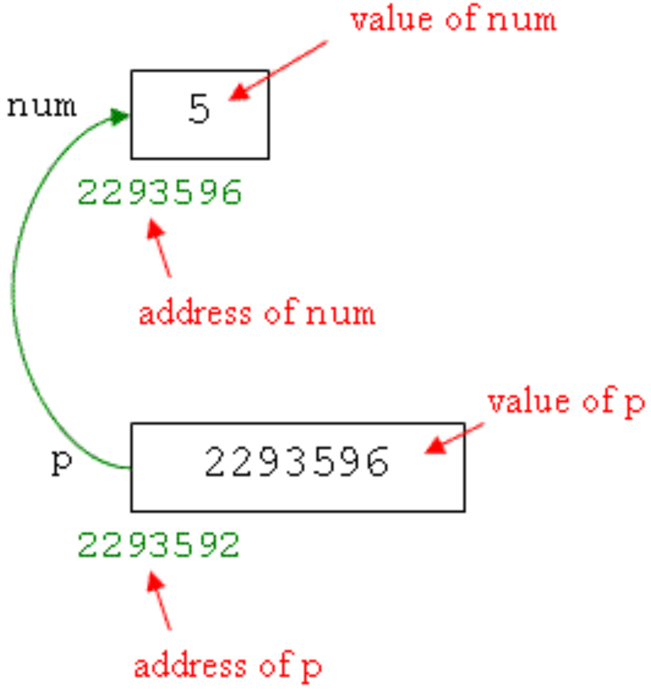


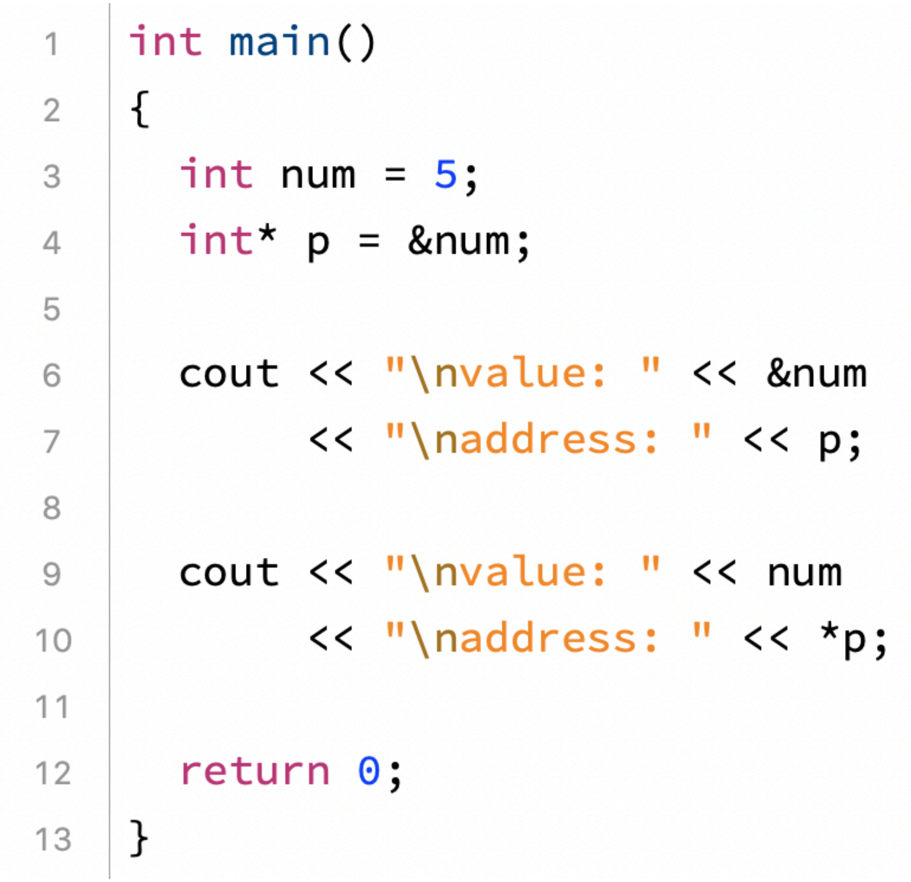
```
int main()
{
  int x = 2500;
  int & friend_of_x = x; // reference

  friend_of_x = 20;

  cout<<"x = "<<x<<endl
      <<"Friend = "<<friend_of_x<<endl;

  return 0;
}
```

Reference Variable (cont)

- pointers are very similar to references, but operate at a lower level
- almost all the mechanics of references are done for you by the compiler
- pointers require you to do the mechanics yourself

Declaring a Pointer





```
int main()
{
   int num = 5;
   int* p = &num;

   cout << "\nvalue: " << &num
        << "\naddress: " << p;

   cout << "\nvalue: " << num
        << "\naddress: " << *p;

   return 0;
}
```

Using Pointer Variable

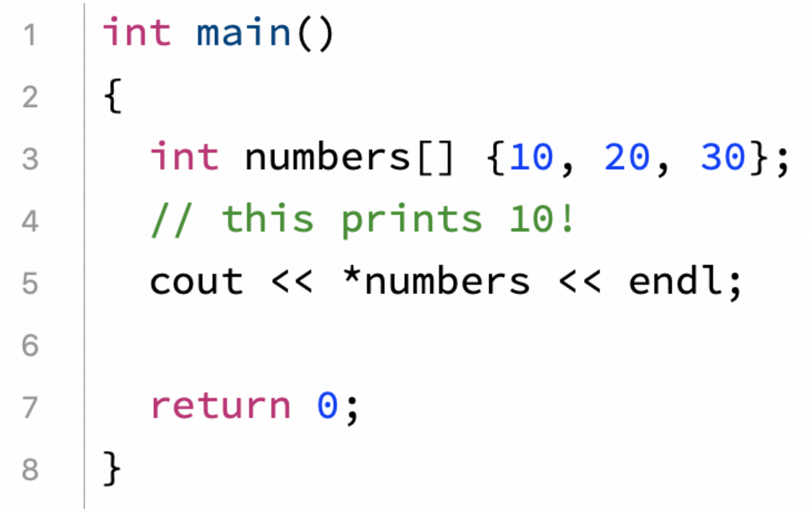
- once a pointer variable has a valid value, it can be used
- the value in the pointer variable itself is an address, usually not directly useful
- to get at the value the pointer is pointing to, we must dereference it using the dereference operator *

Initializing Pointer Variable

- the rules of pointer declaration and initialization are no different than for any other variable
  - declare a variable as close to the point of use as possible
  - initialize a variable at declaration if necessary and useful

Pointers and Arrays

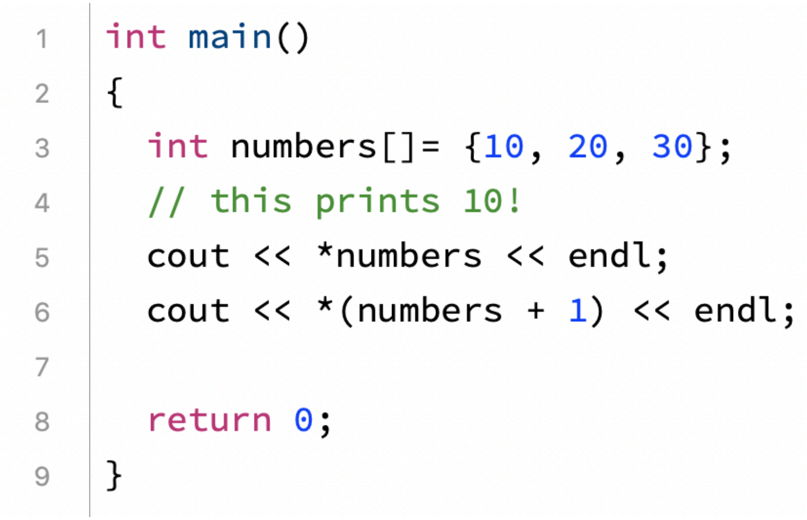
- when we introduced arrays, we said that the array variable name itself, without brackets, really stored the starting address of the array
- but that is exactly what a pointer is!
- an array name is a pointer. Lets see an example to verify that



```cpp
int main()
{
  int numbers[] {10, 20, 30};
  // this prints 10!
  cout << *numbers << endl;

  return 0;
}
```

Pointers and Arrays (cont)

- remember, numbers refers to the address of a byte of memory
- but numbers + 1 does not refer to the byte after numbers
- the compiler knows that an int takes up 4 bytes
- thus "numbers + 1" is really "numbers plus enough bytes to get to the next int"
    - in other words, "numbers plus sizeof int"

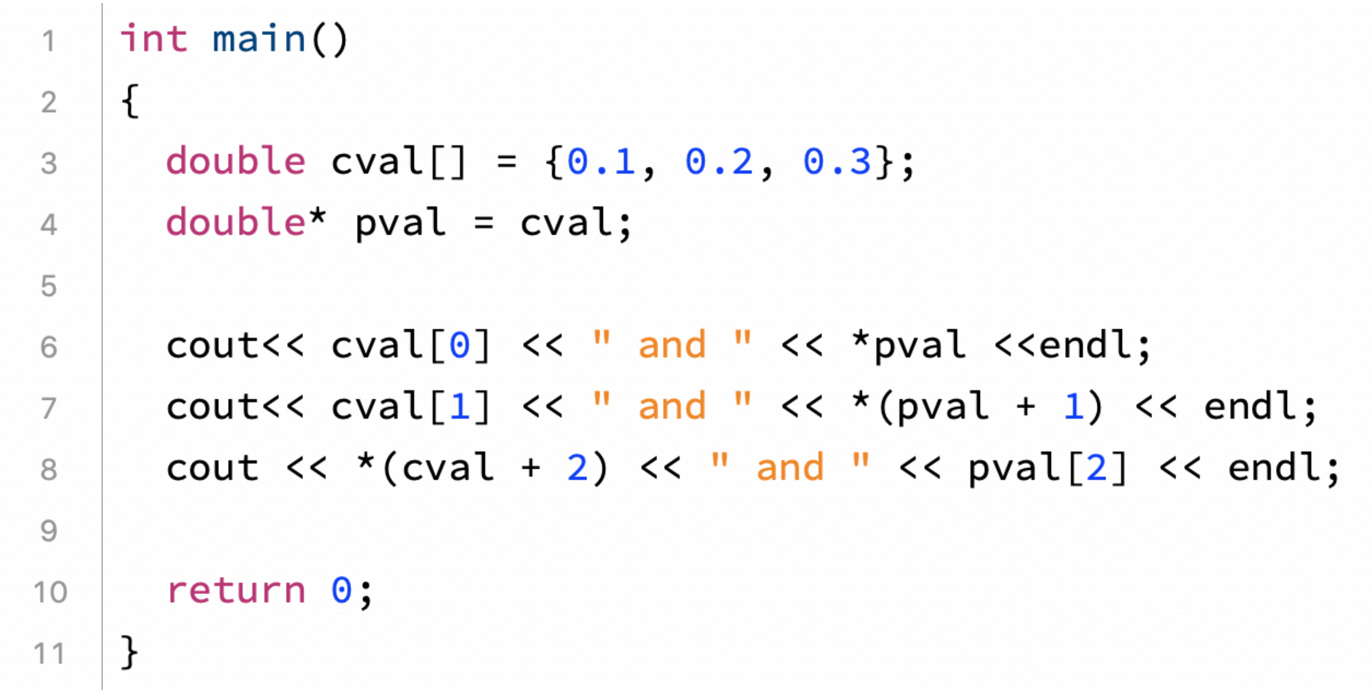


```
int main()
{
  int numbers[]= {10, 20, 30};
  // this prints 10!
  cout << *numbers << endl;
  cout << *(numbers + 1) << endl;

  return 0;
}
```

Syntactic Sugar

- values[index] and *(values + index)
    - are exactly the same thing

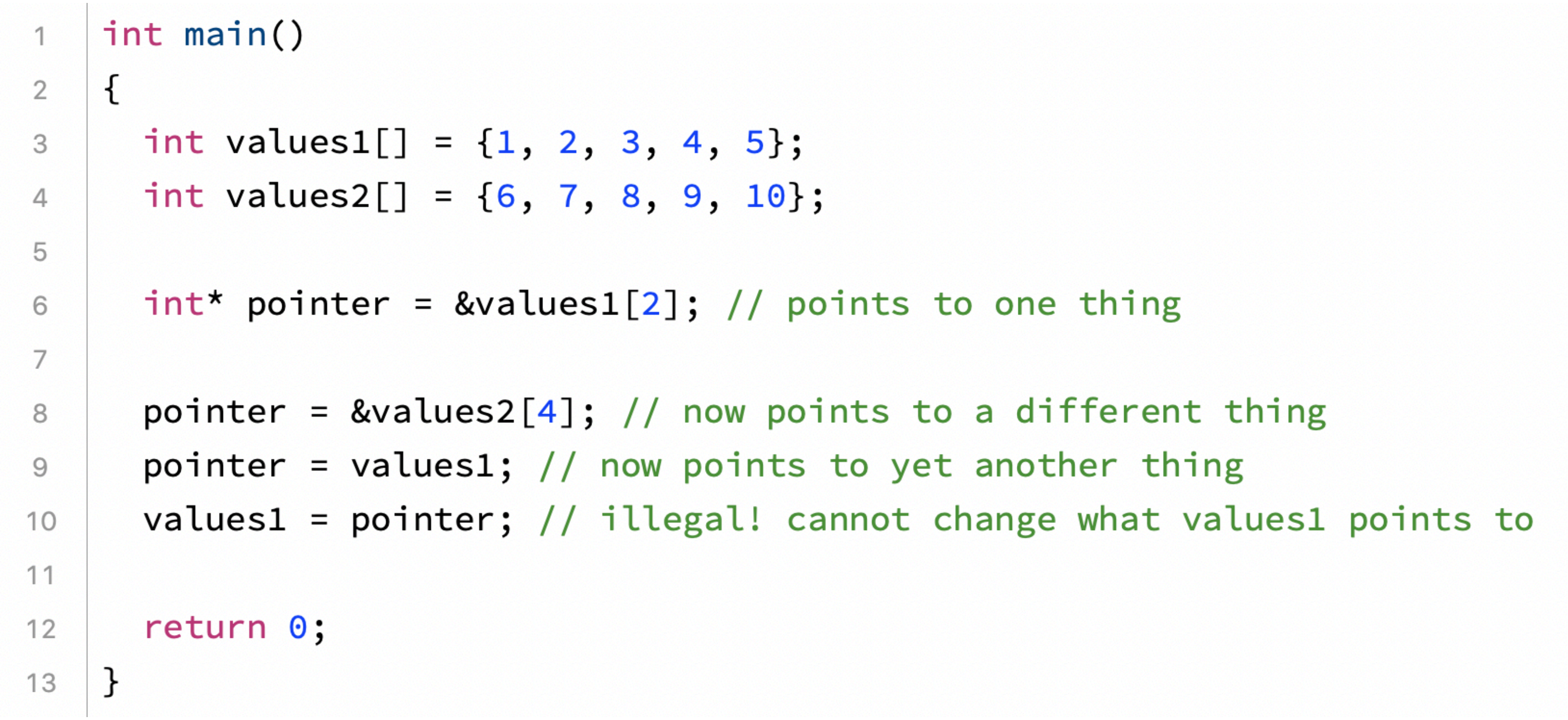
Pointers and Arrays (cont)



```
int main()
{
  double cval[] = {0.1, 0.2, 0.3};
  double* pval = cval;

  cout<< cval[0] << " and " << *pval <<endl;
  cout<< cval[1] << " and " << *(pval + 1) << endl;
  cout << *(cval + 2) << " and " << pval[2] << endl;

  return 0;
}
```

- array names and pointers are interchangeable
- each cout above prints two identical values

Pointers and Arrays (cont): difference



```cpp
int main()
{
  int values1[] = {1, 2, 3, 4, 5};
  int values2[] = {6, 7, 8, 9, 10};

  int* pointer = &values1[2]; // points to one thing

  pointer = &values2[4]; // now points to a different thing
  pointer = values1; // now points to yet another thing
  values1 = pointer; // illegal! cannot change what values1 points to

  return 0;
}
```

- a pointer can be reassigned to point to different things, but an array name cannot be reassigned
  - hence, an array name is a constant pointer

Questions?