The Need for Dynamic Sizing

- Fixing the size of an array “at compile time” is a severe limitation on the efficiency of a program, on its usefulness, or both.

- Think back to the prime numbers example based on the Sieve of Eratosthenes. The maximum array size determines the largest number the program can possibly test. The actual set of numbers to test, specified by the user, must be less than this.

- We would like to have a way of specifying the array size after compilation and after the user input specifies the largest number s/he would like to test.

- C++ provides a mechanism for doing this through dynamic memory allocation and pointers.

The Primes Example

A number of the mechanisms for dynamic memory allocation are illustrated in a revision of the all_primes_sieve program. The code is attached to this handout and will be discussed in detail during lecture.

- No MAX_N is specified

- all_primes_sieve is now a pointer

- It will point to the beginning of an array allocated by the new operator.

- The rest of the program is largely unchanged, including the array subscripting operations.

- Only at the very end of the program is the dynamically allocated array deleted.
This Week

- Pointer variables and memory addresses
- Accessing variables through pointers
- Dynamic allocation
- Pointers and arrays
- Stepping through arrays using pointers
- Pointer arithmetic
- Pointers as function call parameters
- Arrays of pointers

Reading

- Carrano and Prichard, pp 163-172
- Deitel and Deitel, Sections 5.1-5.5, 5.7-5.9
Another example

Consider the following code snippet:

```c
float x = 15.5;
float *p;
p = &x;
*p = 72;
if ( x > 20 )
    cout << "Bigger\n";
else
    cout << "Smaller\n";
```

The output is

Bigger

because \( x == 72.0 \). What’s going on?

**Pointer Variables and Memory Access**

- \( x \) is an ordinary integer, but \( p \) is a pointer that can hold the memory address of an integer.

- The difference is explained in the following picture:
• Every variable is attached to a location in memory. This is where the value of that variable is stored. Hence, we draw a picture with the variable name next to a box that represents the memory location.

• Each memory location also has an address, which is itself just an index into the giant array that is the computer memory.

• The value stored in a pointer variable is an address in memory. In this case, the statement

  \[ p = \&x; \]

  Takes the address of \( x \)'s memory location data is stored and stores it (the address) in the memory location associated with \( x \).

• Since the value of this address is much less important than the fact that the address is \( x \)'s memory location, we depict the address with an arrow.

• The statement

  \[ *p = 72; \]

  cause the computer to get the memory location stored at \( p \), then go to that memory location, and store 72 there. This writes the 72 in \( x \)'s location.

• It will be very (VERY) important to distinguish between \( p \) and \( *p \). Can you do it?

**Defining Pointer Variables**

```c
int * p, q;
float *s, *t;
```

• Here, \( p, s \) and \( t \) are all pointers, but \( q \) is NOT. You need the * before each variable name.

• There is no initialization of pointer variables in this two-line snippet, so a statement of the form

  \[ *p = 15; \]

  will cause some form of “memory exception”.

4
Operations on Pointers

- The unary operator * in the expression *p is the “dereferencing operator”. It means “follow the pointer”

- The unary operator & in the expression &x means “take the memory address of.”

- Pointers can be assigned. We will look at the following example in detail.

```c
float x=5, y=9;
float *p = &x, *q = &y;
*p = 17.0;
*q = *p;
q = p;
*q = 13.0;
```

What are the values of x and y at the end?

- Assignments mixing integers and assignments mixing pointers of different types are illegal. Continuing with the above example:

```c
int *r;
r = q;       // Illegal: different pointer types;
p = 35.1;    // Illegal: float assigned to a pointer
```

- Comparisons between pointers of the form

```c
if ( p == q )
```

or

```c
if ( p != q )
```

are legal and very useful! Less than and greater than comparisons are generally less useful (although allowed).
Null Pointers

- Pointers that don’t point anywhere useful should be given the value 0. This is a legal pointer value.
  - Many compilers define NULL to be a special identifier equal to 0.

- Comparing a pointer to 0 is very useful. It indicates whether or not a pointer has a legal address. For example,

  ```
  if ( p != 0 )
    cout << *p << endl;
  ```

- Dereferencing a null pointer leads to memory exceptions.

Static and Dynamic Memory

- Statically allocated memory is memory defined at compile-time (really, when the program is written).
  - Every program we have written until today has used static allocation only.

- Dynamic allocation occurs when a program asks for new memory during execution (“on-the-fly”) based on information such as a user specifying the number of integers to test for primality.

- Dynamic memory is
  - created using the `new` operator,
  - access through pointers, and
  - removed through the `delete` operator.

- Here’s a simple example involving dynamic allocation of integers:

  ```
  int * p = new int;
  *p = 17;
  cout << *p << endl;
  delete p;
  p = new int;
  *p = 27;
  delete p;
  ```
This example is very much like our earlier examples using pointers to floats except that there is no explicitly named variable other than the pointer variable.

Pointers and Arrays

Pointers are most useful pointing to dynamically allocated arrays and dynamically allocated objects. We’ll re-examine the major steps of the primes example:

```c
bool * is_prime_sieve;
is_prime_sieve = new bool[ n+ 1 ];
...
delete [] is_prime_sieve;
```

- The bracket operator [] is used in both the new (allocation) and delete (deallocation). This is VERY important.
- The pointer `is_prime_sieve` points to the start of the dynamically allocated array.
- You can’t tell from the definition of `is_prime_sieve` whether it will point to an array or to an individual variable.
  - which is why [] is used.
- After dynamic allocation, the pointer variable `is_prime_sieve` is treated as though it is an ordinary array!
- In fact, pointers and array names are in many ways interchangeable in C and C++.

Review

In mentally organizing the material from today, think about the following major topics:

- Dynamic memory is needed to make programs more flexible.
- Pointers are variables that store memory locations of other variables.
- Pointers can be deferred, assigned and compared.
- Dynamic memory is allocated and deallocated (through `new` and `delete`) based on needs of the program at run-time.
• Pointers to dynamically allocated arrays are especially useful.