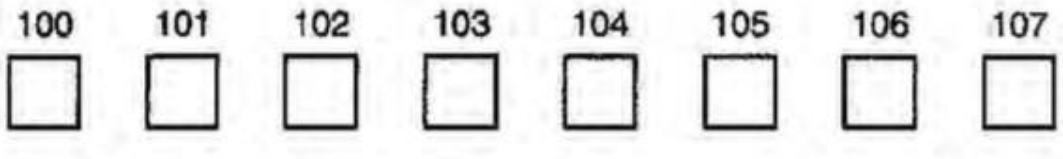
Pointers

Spring 2022

Memory and addresses

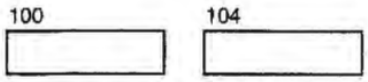
- Each location in memory is identified by a unique address
- Each location in memory contains a value
- Each of these locations is called a byte
  - Modern machines have 8-bit bytes, which can store unsigned integers from [0, 255] or signed integers from [-128, 127].



Memory locations on a real machine

Memory and addresses (cont)

- To store even larger values, we take two or more bytes and treat them as if they were a single, larger unit of memory.
  - For example, many machines store integers in words (2 bytes, e.g. a `short`) or quads (4 bytes, e.g., an `int`).



Two `int`s in adjacent memory

- Because they contain more bits, each word (short) or quad (int) can store a larger range of values.
- Note that even though a quad contains four bytes, it has only one address.

Address versus contents

- Contents of five values in memory

100	104	108	112	120
112	-1	1078523331	100	108

- Variable identifiers substituted for addresses

a	b	c	d	e
112	-1	1078523331	100	108

Values and their types

- Here are the declarations for those variables:

```
int    a = 112;
int    b = -1;
float  c = 3.14;
int*   d = &a;
float* e = &c;
```

Values and their types (cont)

- Variables contain a sequence of bits, `0`'s and `1`'s, that can be interpreted as any data type depending upon the manner in which they're used.
- The type of a value cannot be determined simply by examining its bits.

Values and their types (cont)

- Consider this 32-bit value:

```
01100111011011000110111101100010
```

  - Can be interpreted multiple ways:

Type	Value(s)
1 32-bit integer	1735159650
2 16-bit integers	26476 followed by 28514
four characters	`glob`
floating-point	`1.116533E24`
machine instructions	`beq .+110; ble .+102`

Contents of a pointer variable

- Contents of memory after initialization

```
int    a = 112;
int    b = -1;
float  c = 3.14;
int*   d = &a;
float* e = &c;
```

a	b	c	d	e
112	-1	3.14	100	108

Indirection operator

- Following a pointer to the location to which is points is called **indirection** or **dereferencing** the pointer.

- Examples using previous declarations

Expression	Value	Type
a	112	int
b	-1	int
c	3.14	float
d	100	int*
e	112	float*
*d	112	int
*e	3.14	float

Uninitialized and illegal pointers

- Very common error

```
int* a;
...
*a = 12;
```

- Where does a point?
  - Illegal address (if you're lucky)
    - `segfault`
  - Legal address (if you're not lucky)
    - Very difficult to find
- Be extremely careful that pointers are initialized before applying indirection to them!

```
int value = 42;
int* a = &value;
*a = 12;
```

`nullptr`

- Modern C++ defines `nullptr` as a pointer constant that does not point to anything.
- Provides a way to specify that a particular pointer is not currently pointing to anything.
  - Example: A function whose job is to search an array for a specific value may return a pointer to the array element that was found, but if no element is found, `nullptr` could be returned instead.
- Illegal to dereference `nullptr`
  - Before dereferencing a pointer, you must ensure it is not `nullptr`
- The use of `NULL` or the integer value `0` is deprecated when initializing pointer variables.

Pointers, indirection, and variables

- What does this do?

```
*&a = 42;
```

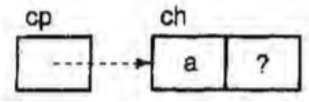
- Functionally equivalent to `a = 42;` but involves more operations, and object code will be larger and slower.

Pointer expressions

- Consider the following declarations

```
char  ch = 'a';
char* cp = &ch;
```

- We now have two variables initialized like this:



The memory location that follows ch is also shown for reference.

Pointer arithmetic

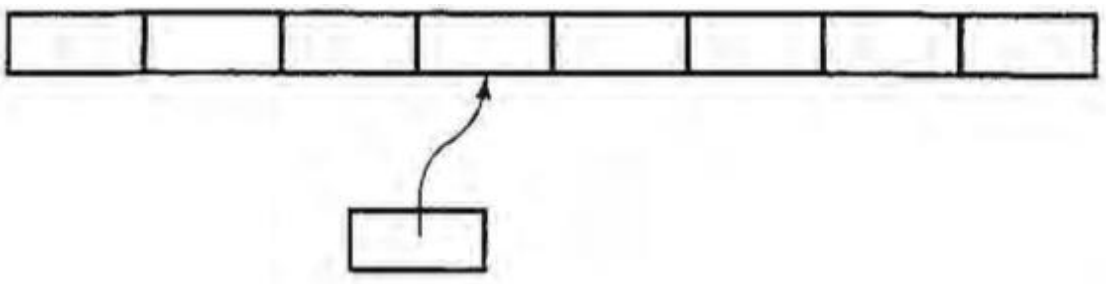
- The result of a pointer plus an integer is another pointer.
- When arithmetic is performed with a pointer and an integer quantity, the integer is *scaled* to the proper size before the addition.
  - The "proper size" is the size of whatever type the pointer is pointing at, and the "scaling" is simply multiplication.

Pointer arithmetic (cont)

Expression	Assuming `cp` is a pointer to a…	… and the size of `*p` is …	Value added to pointer
`p + 1`	`char`	1	1
`p + 1`	`short`	2	2
`p + 1`	`int`	4	4
`p + 1`	`double`	8	8
`p + 2`	`char`	1	2
`p + 2`	`short`	2	4
`p + 2`	`int`	4	8
`p + 2`	`double`	8	16

Arithmetic operations

- Pointer arithmetic in C++ is restricted to exactly two forms.
  - **pointer ± integer**
    - Only valid for pointers that are pointing at an element of an array



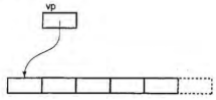
- Undefined if the result points to anything before the first element or after the last element.

Arithmetic operations (cont)

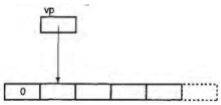
```
const size_t CAPACITY = 5;
float  values[CAPACITY];
float* vp = nullptr;

for (vp = &values[0]; vp < &values[CAPACITY]; ++vp) {
    *vp = 0;
}
```

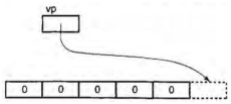
Arithmetic operations (cont)



vp points to first element of array



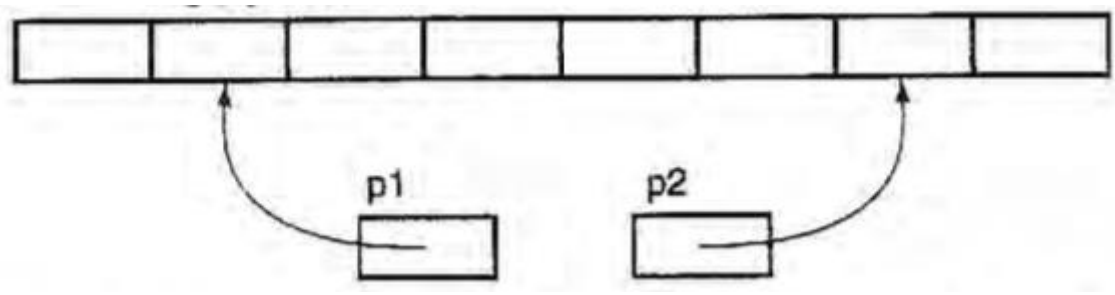
After first iteration of loop



After five iterations of the loop

Arithmetic operations (cont)

- **pointer - pointer**
  - Only allowed when both pointers point to elements of the same array.



- Result of subtracting two pointers is of type `ptrdiff_t`, which is a signed integral type. Value is the distance (measured in array elements, not bytes) between the two pointers
- Assuming `vp` points the address just past the end of the array as before, we can compute the number of elements in the array

```
ptrdiff_t num_elements = vp - &values[0];  // == CAPACITY
```

Relational operations

- Possible to compare two pointer values with relational operators `<`, `<=`, `>`, and `>=` *only if they point to elements of the same array*.
- You may test for equality or inequality between pointers.

Array names

- Consider these declarations

```
int a;
int b[10];
```

- Variable `a` is a scalar data type because it is a single value. The type is `int`.
- Variable `b` is an array, because it is a collection of values. Each element of the array is a scalar.
  - A subscript is used with the array name to identify one specific value from that collection, e.g., `b[0]` identifies the first value in the `b` array.

Array names (cont)

- The type of `b[4]` is an `int`, but what is the type of `b`? To what does it refer?
  - Does not refer to the whole array.
- `b` is a **pointer constant** that is the address of the first element of the array
  - Generated by the compiler when the array name is used in an expression.
- The value of `b` is a pointer constant as opposed to a pointer variable; you cannot change the value of a constant.
  - The value points to where the array begins in memory, so the only way to change it is to move the array somewhere else in memory. Arrays are managed by the compiler, hence the arrays cannot be moved, and the values of array names are pointer constants.

Subscripts

- In the context of the previous declarations, what is the meaning of this expression?

```
*(b + 3)
```

- The value of `b` is a pointer to an integer, so the value of 3 is "scaled" to the size of an `int` (3 * 4 = 12 bytes).
- The addition yields a pointer to the integer that is located three integers beyond the first one in the array.
- The indirection then takes us to this new location, either to get the value there or to store a new one.
- The subscript is exactly the same as an indirection. The following expressions are equivalent

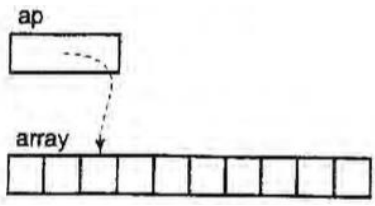
```
array[subscript]
*(array + subscript)
```

Subscripts (cont)

- Consider the following declarations

```
int array[10];
int* ap = array + 2;
```

- Remember "scaling" is performed on the addition, resulting in `ap` pointing to `array[2]`



Subscripts (cont)

Expression	Result
`ap`	This one is easy. You can just read the answer from the initialization: `array + 2`. The expression `&array[2]` is equivalent.
`*ap`	Another easy one. The indirection follows the pointer to the location it is pointing at, which is `array[2]`. You could also write `*(array + 2)`.
`ap[0]`	"You can't do that! `ap` isn't an array!" Remember, a subscript in C++ is exactly like an indirection expression, so you can use one anywhere you can use indirection. In this case, the equivalent expression is `*(ap + 0)`, which, after getting rid of the zero ends up being identical to the previous expression. Therefore, it has the same answer, `array[2]`.

Subscripts (cont)

Expression	Result
`ap + 6`	If `ap` points to `array[2]`, this addition gives a pointer to the location six integers later in the array, which is equivalent to `array + 8` or `&array[8]`.
`*ap + 6`	Be careful, there are two operators here. Which goes first? The indirection. The result of the indirection is then added to 6, so this expression is the same as `array[2] + 6`.
`*(ap + 6)`	The parentheses force the addition to go first, so this time we get the value of `array[8]`. Observe that the indirection used here is in exactly the same form as the indirection of a subscript.
`ap[6]`	Convert the subscript to its indirection form, and you'll see the same expression we just did, so it has the same answer.

Subscripts (cont)

Expression	Result
`&ap`	This expression is perfectly legal, but there is no equivalent expression involving `array` because you cannot predict where the compiler will locate `ap` relative to `array`.
`ap[-1]`	A negative subscript!? A subscript is just an indirection expression. Convert it to that form and then evaluate it. `ap` points to the third element (the one whose subscript is 2), so using an offset of -1 gets us the *previous* element, `array[1]`.
`ap[9]`	This one looks fine but is actually a problem. The equivalent expression is `array[11]`, but there are only ten elements in the array. Evaluating the subscript produces a pointer expression that runs off the end of the array, making this illegal.

Subscripts (cont)

- Here's an interesting tangent. In the context of the previous declarations, what is the meaning of this expression?

```
2[array]
```

- It is legal. Convert it to the equivalent indirection expression and you will see its validity:

```
*(2 + array)
```

- Addition is commutative, so this expression has exactly the same meaning as

```
*(array + 2)
```

Pointers versus subscripts

- If indirection expressions and subscripts are interchangeable, which should you use?

  - Subscripts are easier for most people to understand, especially with multidimensional arrays.
  - Pointers may be more efficient.
- Assuming they are both used correctly, subscripts are never more efficient than pointers, but pointers are frequently more efficient than subscripts.

When pointers are more efficient than subscripts

- Loop 1

```
int array[10];
for (int i = 0; i < 10; ++i) {
    array[i] = 0;
}
```

  - To evaluate the subscript, the compiler replaces with an indirection expression, which takes both space and time as the offset into the array must be recalculated with each iteration.
- Loop 2

```
int  array[10];
for (int* ap = array; ap < array + 10; ++ap) {
    *ap = 0;
}
```

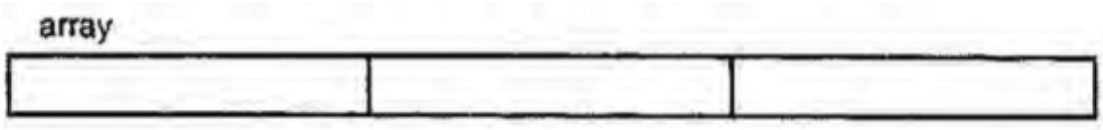
  - On most machines, the second loop is smaller and faster.

Storage order

- Consider this array

```
int array[3];
```

  - It contains three elements as shown.

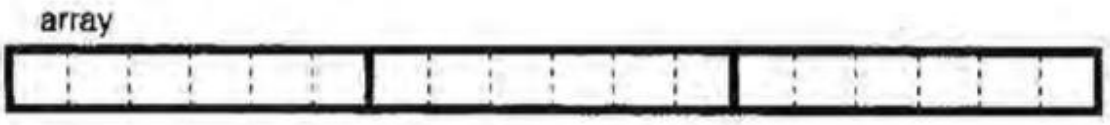


Storage order (cont)

- But now suppose you are told that each of the elements is in fact an array of six elements? Here is the new declaration:

```
int array[3][6];
```

  - and here is how it looks:



- The solid boxes show the three elements of the first dimension, and the dotted divisions show the six elements of the second dimension.
- Denotes the **storage order** of array elements. C++ uses **row major order**.

Arrays of pointers

- Aside from its type, a pointer variable is like any other variable. Just as you can create arrays of integers, you can also declare arrays of pointers. Here is an example:

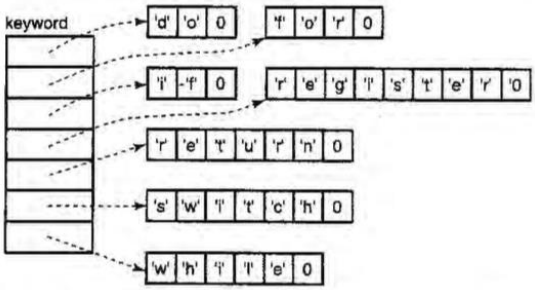
```
int* api[10];
```

- Declares `api` as array 10 of pointer to `int`

Arrays of pointers (cont)

- Example

```
const char* keywords = {
    "do", "for", "if", "register", "return", "switch"
};
```



Summary

- Each location in the computer's memory is identified by an address.
- Adjacent locations are often grouped together to allow larger ranges of values to be stored (e.g., 4 bytes for an `int`).
- A pointer is a value that represents a memory address.
- Neither you nor the computer can determine the type of a value by examining its bits; the type is implicit in how the value is used. The compiler and language semantics ensure that values are used in a way that is appropriate to how they are declared.

Summary (cont)

- Indirection must be applied to a pointer to obtain the value that it points to. The result of applying indirection to a "pointer to an int" is an `int`.

- Declaring a pointer variable does not automatically allocate memory in which to store values!
- Indirection on uninitialized pointer variable is undefined. Pointers variables must always store a valid address or `nullptr`.
- `nullptr` is a pointer constant that points to nothing.

Summary (cont)

- Limited arithmetic can be performed on pointer values. Both pointers must point to elements within the same array to be valid.
  - You can add an integer value to a pointer, and you can subtract an integer value from a pointer.
  - You can subtract one pointer from another.
- Pointers may be compared to each other for equality or inequality.
- Two pointers that point to elements of the same array may also be compared using the relational operators `<`, `<=`, `>`, and `>=` to determine their positions in the array relative to each other.
- The result of relational comparison between unrelated pointers is undefined.

Summary (cont)

- Value of an array name is a pointer to the first element of the array.
- Except for their precedence, the subscript expression `array[value]` is the same as the indirection expression `*(array + value)`.
- Pointer expressions are frequently more efficient than subscripts.
- Pointers and arrays are not equivalent. Declaring an array creates space to store the array elements, while declaring a pointer variable only creates space to store the pointer (an address).

Cautions

- Set pointer variables to `nullptr` when they are not pointing to anything useful.
- Never dereference `nullptr`.