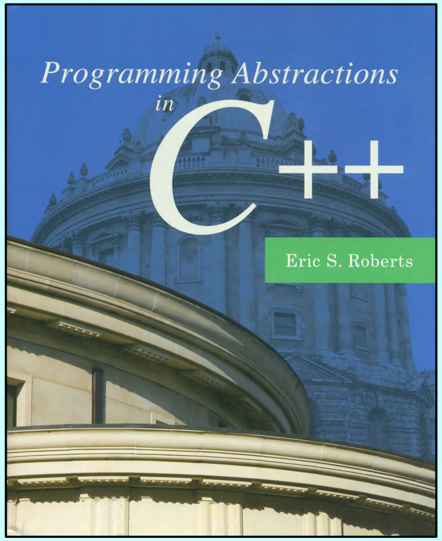
CHAPTER 11

Pointers and Arrays

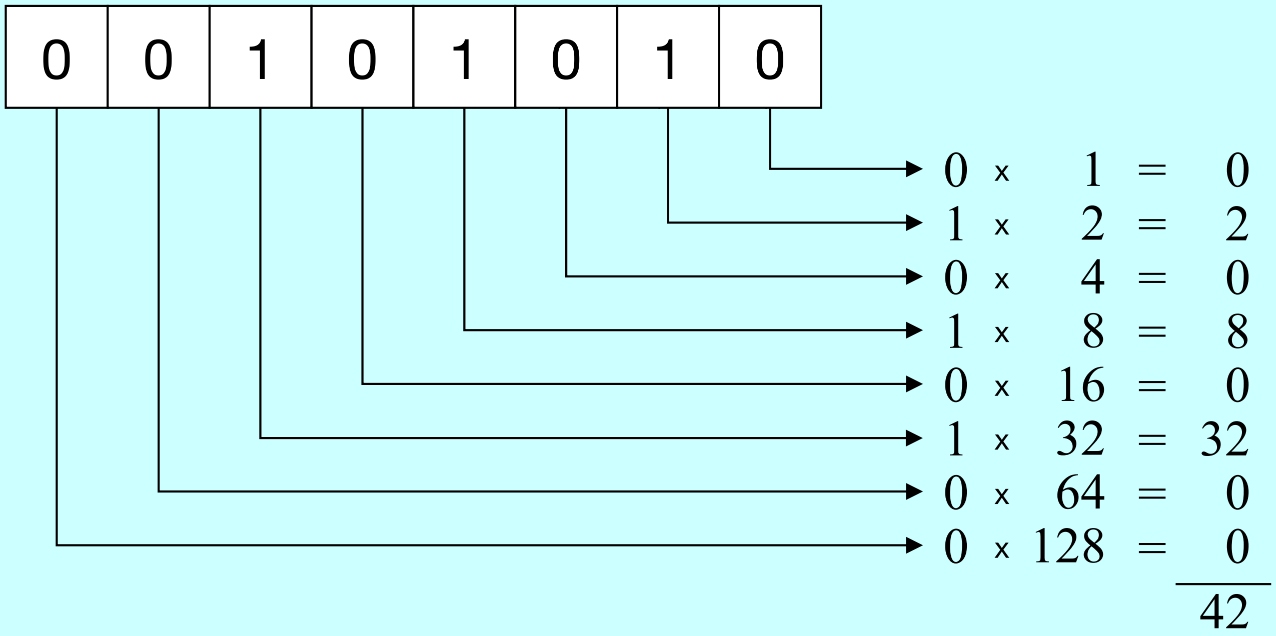
Orlando ran her eyes through it and then, using the first finger of her right hand as pointer, read out the following facts as being most germane to the matter. .

—Virginia Woolf, *Orlando,* 1928



Binary Notation

- Bytes and words can be used to represent integers of different sizes by interpreting the bits as a number in ***binary notation***.
- Binary notation is similar to decimal notation but uses a different ***base***. Decimal numbers use 10 as their base, which means that each digit counts for ten times as much as the digit to its right. Binary notation uses base 2, which means that each position counts for twice as much, as follows:



Numbers and Bases

- The calculation at the end of the preceding slide makes it clear that the binary representation 00101010 is equivalent to the number 42. When it is important to distinguish the base, the text uses a small subscript, like this:

$$00101010_2 = 42_{10}$$

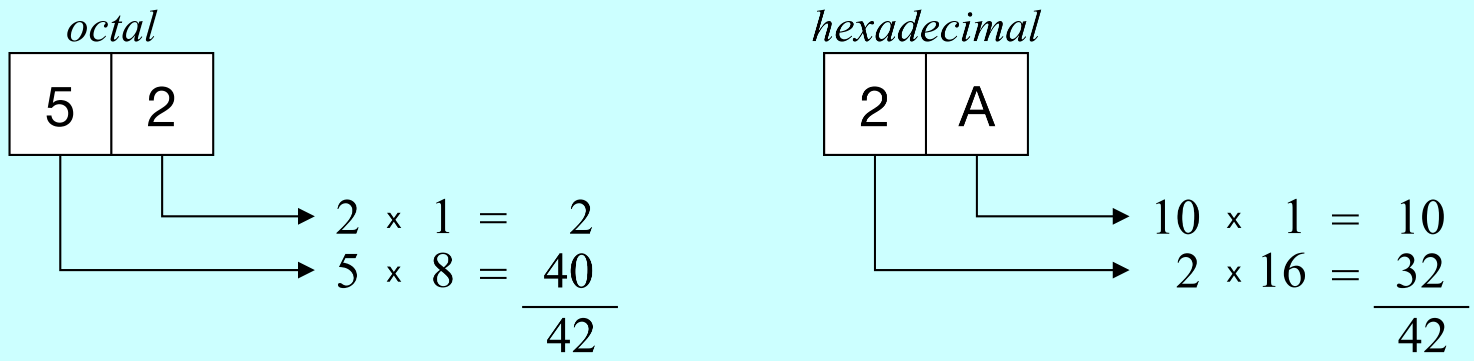
- Although it is useful to be able to convert a number from one base to another, it is important to remember that the number remains the same. What changes is how you write it down.
- The number 42 is what you get if you count how many stars are in the pattern at the right. The number is the same whether you write it in English as *forty-two,* in decimal as 42, or in binary as 00101010.

- Numbers do not have bases; representations do.

Octal and Hexadecimal Notation

- Because binary notation tends to get rather long, computer scientists often prefer ***octal*** (base 8) or ***hexadecimal*** (base 16) notation instead. Octal notation uses eight digits: 0 to 7. Hexadecimal notation uses sixteen digits: 0 to 9, followed by the letters A through F to indicate the values 10 to 15.

- The following diagrams show how the number forty-two appears in both octal and hexadecimal notation:



- The advantage of using either octal or hexadecimal notation is that doing so makes it easy to translate the number back to individual bits because you can convert each digit separately.

Exercises: Number Bases

- What is the decimal value for each of the following numbers?

10001_2	177_8	AD_{16}
17	127	173

- As part of a code to identify the file type, every Java class file begins with the following sixteen bits:

1	1	0	0	1	0	1	0	1	1	1	1	1	1	1	0

- How would you express that number in hexadecimal notation?

1	1	0	0	1	0	1	0	1	1	1	1	1	1	1	0

$CAFE_{16}$

The Structure of Memory

- The fundamental unit of memory inside a computer is called a ***bit***, which is a contraction of the words *binary digit*. A bit can be in either of two states, usually denoted as 0 and 1.
- The hardware structure of a computer combines individual bits into larger units. In most modern architectures, the smallest addressable unit on which the hardware operates is a sequence of *eight consecutive bits* called a ***byte***. The following diagram shows a byte containing a combination of 0s and 1s:

0	0	1	0	1	0	1	0

- Numbers are stored in still larger units that consist of multiple bytes. The unit that represents the most common integer size on a particular hardware (i.e., the number of bits the CPU can process at one time) is called a (hardware) ***word***. Because machines have different architectures, the number of bits in a word may vary from machine to machine. E.g., a word in *x86-64*, the 64-bit version of the x86 instruction set, is 64-bit.

Word Size and Address Length

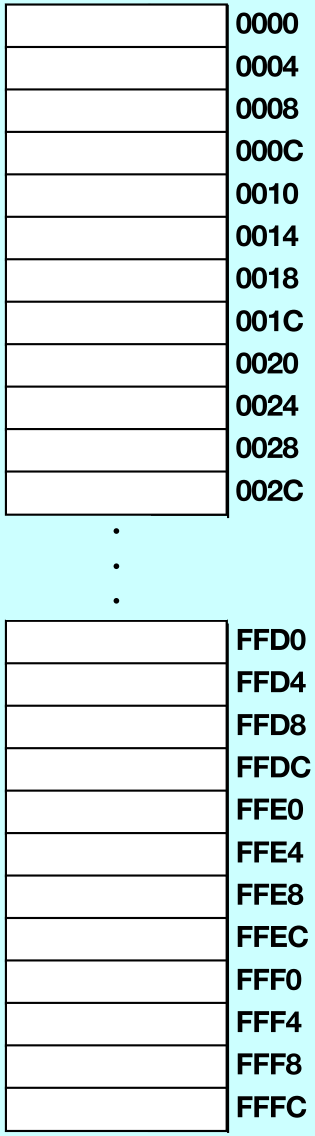
- A word is usually the largest piece of data that can be transferred to/from the memory in a single operation of a particular processor, so the *word size* is an important characteristic of any specific processor/architecture.
- The largest possible *address length*, used to designate a location in memory, is typically a word, because this allows one memory address to be efficiently stored in one word.
- Very often, when referring to the *word size* of a modern computer, one is also describing the *address length* on that computer. E.g., a computer said to be "32-bit" has a hardware word size of 32 bits, and also usually allows 32-bit memory addresses.
- However, this does not always hold true. Computers can have memory addresses larger or smaller than their word size.

Address Length and Memory Size

- Although we will make up some four-digit hexadecimal (i.e., 16-bit) numbers as the memory addresses for simplicity in our examples, the actual address lengths may be different on different computers/architectures.
- In theory, modern byte-addressable N-bit computers can address 2^N bytes of memory, but in practice the amount of memory is limited by the CPU, the memory controller, etc.
- Exercise: The theoretical memory sizes in 16-, 32-, and 64-bit machines are?
  - 16 bit → 65, 536 bytes (64 Kilobytes)
  - 32 bit → 4, 294, 967, 295 bytes (4 Gigabytes)
  - 64 bit → 18, 446, 744, 073, 709, 551, 616 (16 Exabytes)

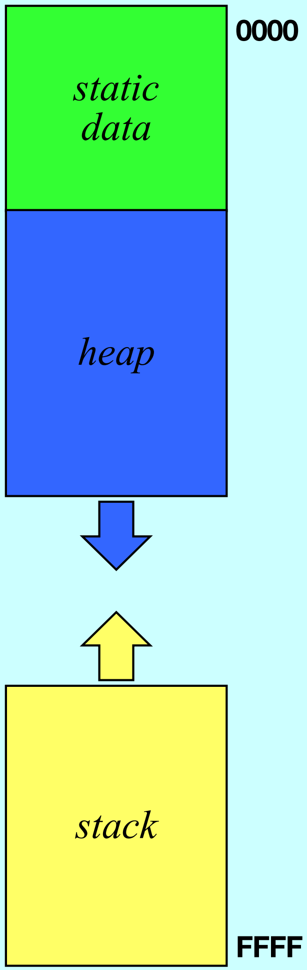
Memory and Addresses

- Every byte inside the primary memory of a machine is identified by a numeric address. The addresses begin at 0 and extend up to the number of bytes in the machine, as shown in the diagram on the right.
- Memory diagrams that show individual bytes are not as useful as those that are organized into words. The revised diagram on the right now includes four bytes (i.e., a 32-bit machine) in each of the memory cells, which means that the address numbers increase by four each time.
- In these slides, addresses are four-digit hexadecimal numbers, which makes them easy to recognize.
- When you create memory diagrams, you don't know the actual memory addresses at which values are stored, but you do know that everything has an address. ***Just make something up.***



The Allocation of Memory to Variables

- When you declare a variable in a program, C++ allocates space for that variable from one of several memory regions.
- One region of memory is reserved for program code and global variables/constants that persist throughout the lifetime of the program. This information is called ***static data***.
- Each time you call a method, C++ allocates a new block of memory called a ***stack frame*** to hold its local variables. These stack frames come from a region of memory called the ***stack***.
- It is also possible to allocate memory dynamically, as we will describe in Chapter 12. This space comes from a pool of memory called the ***heap***.
- In classical architectures, the stack and heap grow toward each other to *maximize the available space*.



Memory Space: an Analogy

Hotel	**Memory**
Bed	Bit
Room	Byte
Floor	Word
Room number	Address
Number of rooms	Memory size
Extended-stay rooms	Static
Rooms sold offline	Heap
Rooms booked online	Stack

Data Types in C

The data types that C++ inherits from C:

- Atomic (primitive) types:
  - `short`, `int`, `long`, and their `unsigned` variants
  - `float`, `double`, and `long double`
  - `char`
  - `bool`
- Enumerated types defined using the `enum` keyword
- Structure types defined using the `struct` keyword
- Arrays of some base type
- Pointers to a target type

Sizes of the Fundamental Types

- The memory space required to represent a value depends on the type of value. Although the C++ standard actually allows compilers some flexibility, the following sizes are typical:

1 byte (8 bits)	2 bytes (16 bits)	4 bytes (32 bits)	8 bytes (64 bits)	16 bytes (128 bits)
`char`	**`short`**	**`int`**	**`long`**	**`long double`**
`bool`		**`float`**	**`double`**	

- Enumerated types are typically assigned the space of an **`int`**.
- Structure types have a size equal to the sum of their fields.
- Arrays take up the element size times the number of elements.
- Pointers take up the space needed to hold an address, which is usually the size of a hardware word, e.g., 4 bytes on a 32-bit machine and 8 bytes on a 64-bit machine.
- **`sizeof(t)`** returns the actual number of bytes required to store a value of the type **`t`**; **`sizeof x`** returns the actual memory size of the variable **`x`**.

Variables

Flashback

- A variable in C++ is most easily envisioned as a box capable of storing a value. For the following statement:

```
int total = 42;
```

(name is) **`total`**

(stores at) FFD0 | 42 | (contains an) **`int`**

- Each variable has the following attributes:
  - A ***name***, which enables you to differentiate one variable from another.
  - A ***type***, which specifies what type of value the variable can contain.
  - A ***value***, which represents the current contents of the variable.
  - For now, let's not worry about the ***address*** first.
- The address and type of a named variable are fixed. The value changes whenever you *assign* a new value to the variable.

Using addresses as data values: lvalue

- In C++, any expression that refers to an internal memory location capable of storing data is called an ***lvalue***, which can appear on the left side of an assignment statement in C++.
- Intuitively, if it cannot be on the left side of an assignment, or if you cannot assign a value to it, it is not an lvalue.
- The following properties apply to lvalues in C++:
  - Every lvalue is stored somewhere in memory and therefore has an address.
  - Once it has been declared, the address of an lvalue never changes, even though the contents of those memory locations may change.
  - The address of an lvalue is a value of a pointer variable, which can be stored in memory and manipulated as data.

Pointers

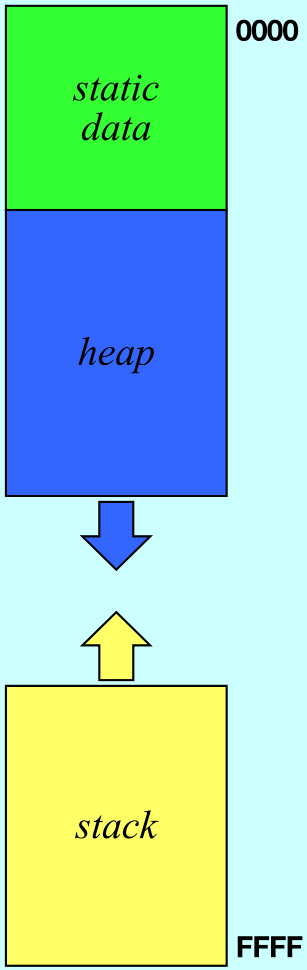
- In C++, every data item is stored somewhere in memory and can therefore be identified with that address. Because C++ is designed to allow programmers to control data at the lowest level, it makes the fact that memory locations have addresses visible to the programmer.
- A data item whose value is an address in memory is called a ***pointer***, which can be manipulated just like any other kind of data. In particularly, you can assign one pointer value to another, which means that the two pointers end up indicating the same data item.
- Diagrams that involve pointers are typically represented in two different ways:
  - Using memory addresses emphasizes the fact that pointers are just like integers.
  - Conceptually, it often makes more sense to represent a pointer as an arrow.

Pointers

- In C++, pointers serve several purposes, of which the following are the most important:
  - *Pointers allow you to refer to a large data structure in a compact way*. Because a memory address typically fits in a few bytes of memory, this strategy offers considerable space savings when the data structures themselves are large. E.g., ***call by pointers***.
  - *Pointers make it possible to reserve new memory during program execution*. In many applications, it is convenient to acquire new memory as the program runs and to refer to that memory using pointers, which is called ***dynamic allocation***.
  - *Pointers can be used to record relationships among data items*. Data structures that use pointers to create connections between individual components are called ***linked structures***. Programmers can indicate that one data item follows another in a conceptual sequence by including a pointer to the second item in the internal representation of the first.

Question: how to design a pointer?

- Imagine that we want to use the memory just like any linear structure, such as a **`Vector`**. Is it possible? And how?
  - Define a variable to represent the address (just like the index **`i`** in **`Vector`**)
  - For each address variable, indicate the element type stored in that address (why?)
  - Provide a way to access the element using the address variable (just like **`[i]`**)
  - Better yet, provide a way to retrieve the address from a regular variable
  - Provide arithmetic operations for the address variables



Declaring a Pointer Variable

- Pointer variables have a declaration syntax that may at first seem confusing. To declare a variable as a pointer to a particular type as opposed to a variable of that type, all you need to do is add a **`*`** in front of the variable name, like this:

```
type*   var;
type *  var;
type   *var;
```

- For example, if you wanted to declare a variable **`px`** to be a pointer to a **`double`** value, you could do so as follows:

```
double * px;
```

- Similarly, to declare a variable **`pptr`** as a pointer to a **`Point`** structure, you would write:

```
Point * pptr;
```

Pointer Operators

- C++ includes two built-in operators for working with pointers:
  - The address-of operator (`&`) is written before a variable name (or any expression to which you could assign a value, an lvalue) and returns the address of that variable.
  - The value-pointed-to operator (`*`) is written before a pointer expression and returns the actual value of a variable to which the pointer points (***dereferencing***).
- Suppose, for example, that you have declared and initialized the following variables:

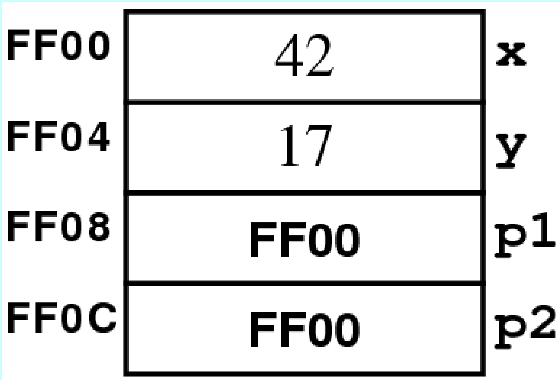
```
double x = 2.5;
double * px = &x;
```

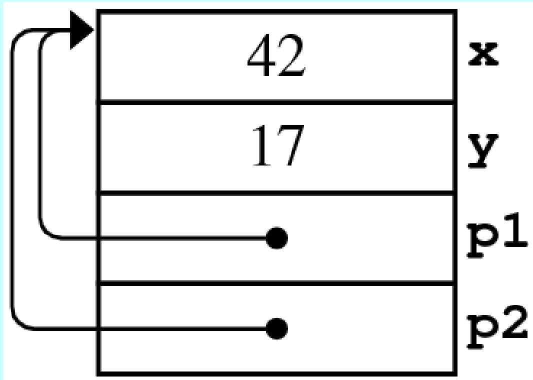
- At this point, the pointer variable **`px`** points to the double variable **`x`**, and the expression **`*px`** is synonymous with the variable **`x`**.

```
double y = *px;
```

Pointer Diagrams

```
int x, y;
int *p1, *p2;
x = 42;
y = 163;
p1 = &y;
p2 = &x;
*p1 = 17;
p1 = p2;
*p1 = *p2;
```





Pointers and Call by Reference

- To swap two integers, the function **`swap`** takes its parameters *by reference*, which means that the stack frame for **`swap`** is given the *addresses* of the calling arguments rather than the *values*.

```
void swap(int & x, int & y) {
   int tmp = x;
   x = y;
   y = tmp;
}
```

```
swap(n1, n2);
```

- You can simulate the effect of call by reference by making the pointers explicit (call by pointer):

```
void swap(int * px, int * py) {
   int tmp = *px;
   *px = *py;
   *py = tmp;
}
```

```
swap(&n1, &n2);
```

Pointer vs. Reference

	Pointer	Reference
Definition	The memory address of an object	An alternative identifier for an object
Declaration	`int i = 5;` `int * p = &i;`	`int i = 5;` `int & r = i;`
Dereferencing	`*p`	
Has an address	Yes (`&p`)	No (the same as `&i`)
Pointing/referring to nothing	Yes (`NULL`/ `nullptr` since C++11)	No
Reassignments to new objects	Yes	No
Supported by	C and C++	C++

The address-of operator, not a reference.

Pointers to Objects

```
Point pt(3, 4);

Point * pp = &pt;
```

- The above code declares two local variables. The variable **`pt`** contains a **`Point`** object with the coordinate values **`3`** and **`4`**. The variable **`pp`** contains a pointer to that same **`Point`** object.
- To invoke the method, e.g., **`getX()`**, of the object:

```
pt.getX();

(*pp).getX();

pp->getX();
```

The -> Operator

- In C++, pointers are explicit. Given a pointer to an object, you need to dereference the pointer before selecting a field or calling a method. Given the definition of **`pp`** from the previous slide, you *cannot* write:

```
pp.getX();
```

because **`pp`** is not a structure or an object of a class.

- You also *cannot* write:

```
*pp.getX();
```

because "." takes precedence over "*". It is equivalent to:

```
*(pp.getX());
```

- To call a method given a pointer to an object, you need to write:

```
(*pp).getX();
```

```
pp->getX();
```

The Keyword `this`

- In the implementation of the methods within a class, you can usually refer to the private instance variables of that class using just their names. C++ resolves such names by looking for matches in the following order (principle of proximity):
  - Parameters or local variables declared in the current method
  - Instance variables of the current object
  - ~~Global variables defined in this scope~~
- It is often convenient to use the same names for parameters and instance variables. If you do, you must use the keyword **`this`** (defined as a pointer to the current object) to refer to the instance variable, as in the constructor for the **`Point`** class (think of **`this`** as **`self`** in Python) :

```
Point::Point(int cx, int cy) {
   x = cx;
   y = cy;
}
```

```
Point::Point(int x, int y) {
    this->x = x;
    this->y = y;
}
```

Simple Arrays in C++

- We have previously used arrays in their low-level form only scarcely so far:

```
char cstr[10];
char cstr[] = "hello";
char cstr[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

  because the **`Vector`** class is so much better.

- From the client perspective, an array is like a brain-damaged form of **`Vector`** with the following differences:
  – The only operation is selection using **`[]`**;
  – Array selection does not check that the index is in range;
  – The declared length of an array is fixed at the time it is created;
  – Arrays don't store their actual length, so programs that use them must pass an extra integer value that represents the number of elements actively in use.

Simple Arrays in C++

- Array variables are declared using the following syntax:

  type name`[`*n*`];`

  where *type* is the element type, *name* is the array name, and *n* is a constant integer expression indicating the length.

- Array variables can be given initial values at the time they are declared:

```
int DIGITS[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

- The size of the array specified in the declaration is called the ***allocated size***. The number of elements actively in use is called the ***effective size***.

- To determine how many elements there are in a strange array (e.g., declared by someone else or dynamically changed):

```
sizeof MY_ARRAY / sizeof MY_ARRAY[0]
```

Pointers and Arrays

- In C++, the name of an array is synonymous with a pointer to its first element. For example, if you declare an array

  ```
  int list[100];
  ```

  the C++ compiler treats the name `list` as a pointer to the address `&list[0]` *whenever necessary*, and `list[i]` is just `*(list+i)`, because pointer arithmetic counts the objects pointed to by the pointer.

- Although an array is often treated as a pointer, they are not entirely equivalent. E.g., you can assign an array to a pointer (of the same type), but not vice versa, because an array is a non-modifiable lvalue (so are constant variables).

- When you pass an array to a function, only the *address* of the array is copied into the parameter. This strategy has the effect of *sharing* the elements of the array between the function and its caller (i.e., call by pointer).

A Simple Array Example

```
const int N = 10;

int main() {
   int array[N];
   for ( int i = 0 ; i < N ; i++ ) {
      array[i] = randomInteger(100, 999);
   }
   sort(array, N);
}
```

This is not an accurate illustration of
`int array[N];`
but more like:
```
int arr[N];
int* array = arr;
```

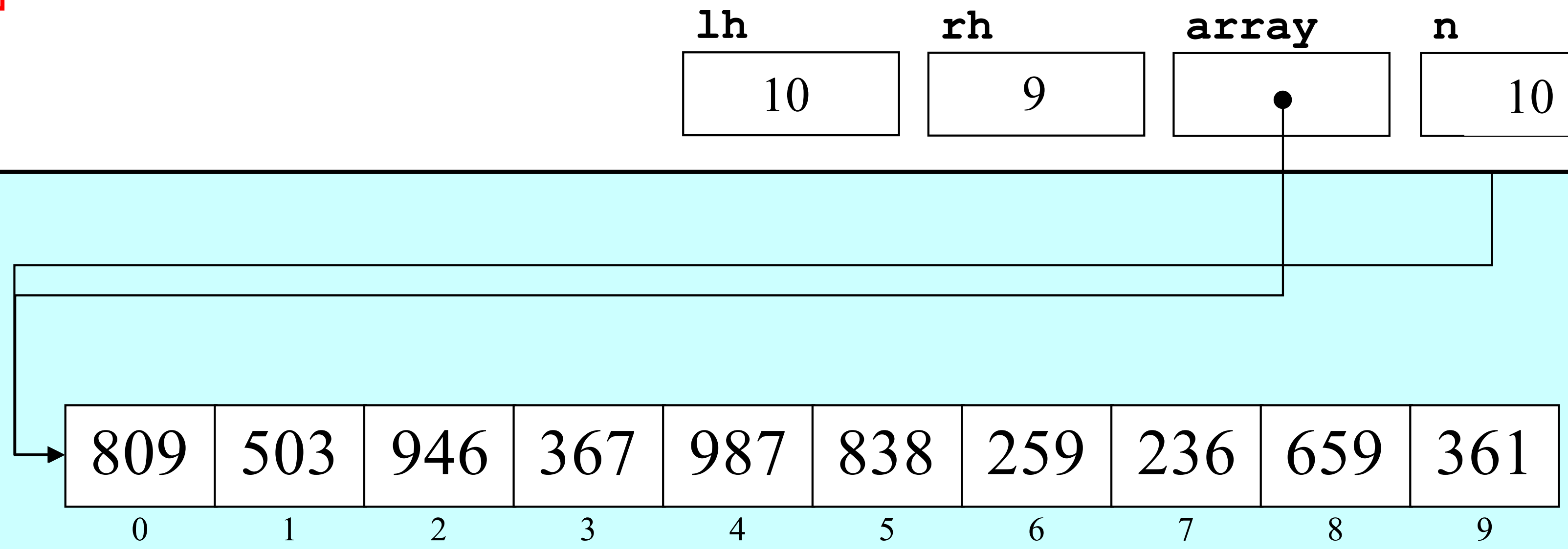
i: 10   **array**: •

809	503	946	367	987	838	259	236	659	361
0	1	2	3	4	5	6	7	8	9

skip simulation

Arrays Are Passed as Pointers

```
void sort(int array[], int n) {
   for ( int lh = 0 ; lh < n ; lh++ ) {
      int rh = findSmallest(array, lh, n - 1);
      swap(array[lh], array[rh]);
   }
}
```



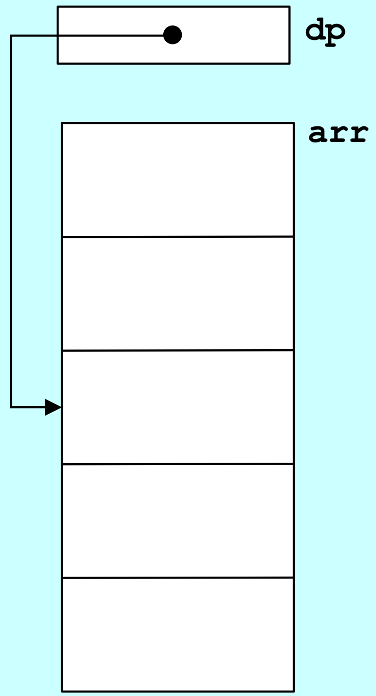
skip simulation

Pointer Arithmetic

- Like C before it, C++ defines the **+** and **-** operators so that they work with pointers. Dangerous, though. Be careful!
- Suppose, for example, that you have made the following declarations:

```
double arr[5];
double * dp = arr;
```

How do those variables appear in memory?



- C++ defines pointer addition so that the following identity always holds (Note the following are not C++ statements!):

```
arr[i] ≡ *(arr+i) ≡ *(dp+i) ≡ dp[i]
&arr[i] ≡ arr+i ≡ dp+i ≡ &dp[i]
```

Thus, **dp + 2** points to **arr[2]**.

Pointers and Arrays

Interpret **a** *as an entire array first. If it doesn't make sense, interpret it as a pointer then.*

```
int a[] = {0, 1, 2, 3};
int * p = a; // &a[0]
```

	`a`	`&a`	`&a[0]`	`p`
Type	array or *used as* pointer to an **`int`**	address of an array	address of an **`int`**	pointer to an **`int`**
Size of	16 (4 **`int`**)	8 (a word)	8 (a word)	8 (a word)
Lvalue	Yes (non-modifiable)	No	No	Yes
Value	**`ADDRESS1`**	**`ADDRESS1`**	**`ADDRESS1`**	**`ADDRESS1`**
`*`	`0`	**`ADDRESS1`**	`0`	`0`
`&`	**`ADDRESS1`**	N/A	N/A	**`ADDRESS2`**
`+1`	**`ADDRESS1 +1 int`**	$\boxed{\texttt{ADDRESS1 +4 int}}$	**`ADDRESS1 +1 int`**	**`ADDRESS1 +1 int`**

C Strings are Pointers to Characters

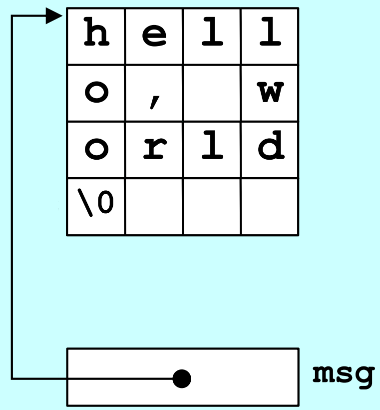
- As you know from Chapter 3, C++ supports the old C style of strings, which is simply a pointer to a character, which is the first element of a character array terminated by the ***null character*** (`'\0'`).
- Given this definition, what does the declaration

```
char* msg = "hello, world";
```

```
char cstr[] = "hello, world";
char* msg = cstr;
```

generate in memory?

- You can still select characters in **`msg`** by their index because of the equivalence of arrays and pointers.

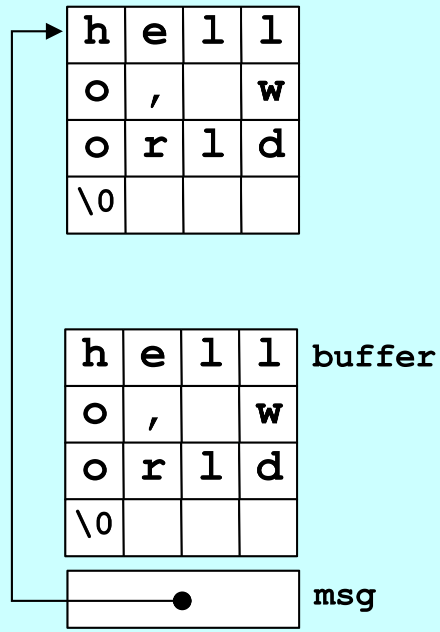


Examples: C String Functions

1. Implement the C library function **`strlen(cstr)`** that returns the length of the C string **`cstr`**.

2. Implement the C library function **`strcpy(dst, src)`**, which copies the characters from the string **`src`** into the character array indicated by **`dst`**. For example, the code on the left should generate the memory state on the right:

```
char* msg = "hello, world";
char buffer[16];
strcpy(buffer, msg);
```



C String Functions `strlen(cstr)`

```
int strlen(char str[]) {
    int n = 0;
    while (str[n] != '\0') {
        n++;
    }
    return n;
}

int strlen(char *str) {
    int n = 0;
    while (*str++ != '\0') {
        n++;
    }
    return n;
}

int strlen(char *str) {
    char *cp;
    for (cp = str; *cp != '\0'; cp++);
    return cp - str;
}
```

It doesn't make sense to add two pointers, however.

`strcpy`: the Hot-Shot Solution

```
void strcpy(char* dst, char* src) {
   while (*dst++ = *src++);
}
```

- The pointer expression **`*p++`** is equivalent to **`*(p++)`**, because unary operators in C++ are evaluated in right-to-left order.
- The **`*p++`** idiom means dereference **`p`** and return as an lvalue the object to which it currently points, and increment the value of **`p`** so that the new **`p`** points to the next element in the array.
- When you work with C++, understanding the **`*p++`** idiom is important primarily because the same syntax comes up in STL iterators, which are used everywhere in professional code.
- It is, however, equally important that you avoid using it in your own code, to avoid ***buffer overflow errors***.

The Internet Worm

"All the News That's Fit to Print"

The New York Times

Late Edition

New York: Today, partly sunny, milder. High 59-64. Tonight, mostly cloudy. Low 48-54. Tomorrow, cloudy, windy, rain developing. High 57-62. Yesterday: High 58, low 41. Details, page D16.

VOL.CXXXVIII... No. 47,679 — NEW YORK, FRIDAY, NOVEMBER 4, 1988 — 35 CENTS

'Virus' in Military Computers Disrupts Systems Nationwide

By JOHN MARKOFF

In an intrusion that raises questions about the vulnerability of the nation's computers, a Department of Defense network has been disrupted since Wednesday by a rapidly spreading "virus" program apparently introduced by a computer science student.

The program reproduced itself through the computer network, making hundreds of copies in each machine it reached, effectively clogging systems linking thousands of military, corporate and university computers around the nation and preventing them from doing additional work. The virus is thought not to have destroyed any files.

By late yesterday afternoon computer experts were calling the virus the largest assault ever on the nation's computers.

'The Big Issue'

"The big issue is that a relatively benign software program can virtually bring our computing community to its knees and keep it there for some time," said Chuck Cole, deputy computer security manager at Lawrence Livermore Laboratory in Livermore, Calif., one of the sites affected by the intrusion. "The cost is going to be staggering."

Clifford Stoll, a computer security expert at Harvard University, added: "There is not one system manager who is not tearing his hair out. It's causing enormous headaches."

The affected computers carry a tremendous variety of business and research information among military officials, researchers and corporations.

While some sensitive military data are involved, the computers handling the nation's most sensitive secret information, like that on the control of nuclear weapons, are thought not to have been touched by the virus.

Parallel to Biological Virus

Computer viruses are so named because they parallel in the computer world the behavior of biological viruses. A virus is a program, or a set of instructions to a computer, that is either planted on a floppy disk meant to be used with the computer or introduced when the computer is communicating over telephone lines or data networks with other computers.

The programs can copy themselves into the computer's master software, or operating system, usually without calling any attention to themselves. From there, the program can be passed to additional computers.

Depending upon the intent of the software's creator, the program might cause a provocative but otherwise harmless message to appear on the computer's screen. Or it could systematically destroy data in the computer's memory. In this case, the virus program did nothing more than reproduce itself rapidly.

The program was apparently a result of an experiment, which

Continued on Page A21, Column 2

PENTAGON REPORTS IMPROPER CHARGES FOR CONSULTANTS

CONTRACTORS CRITICIZED

Inquiry Shows Routine Billing of Government by Industry on Fees, Some Dubious

By JOHN H. CUSHMAN Jr.

Special to The New York Times

WASHINGTON, Nov. 3 — A Pentagon investigation has found that the nation's largest military contractors routinely charge the Defense Department for hundreds of millions of dollars paid to consultants, often without justification.

The report of the investigation said that neither the military's current rules nor the contractors' own policies are adequate to assure that the Government does not improperly pay for privately arranged consulting work. Senior Defense Department officials said the Pentagon was proposing changes to correct the flaws.

While it is not improper for military contractors to use consultants in performing work for the Pentagon, the work must directly benefit the military if it is to be paid for by the Defense Department. Often, Pentagon investigators discovered, this test is not met.

Broader Look at Consultants

The Justice Department's continuing criminal investigation has focused attention on consultants and their role in the designing and selling of weapons, and the Defense Department has been criticized for using consultants too freely. Now the Pentagon's own inves-

Gov. Michael S. Dukakis having his picture taken by a 10-year-old fan at a town meeting in Fairless Hills, Pa., during a tour of the Northeast in which he emphasized the drug problem. Page A10. Vice President Bush addressed supporters a rally in Columbus, Ohio. Less than a week after Mr. Dukakis acknowledged being a liberal, Mr. Bush said yesterday that "this election is not about labels." Page A18.

Registration Off Since 1984 Vote

There has been a pronounced decline in the percentage of eligible Americans who are registered to vote, a research group reports.

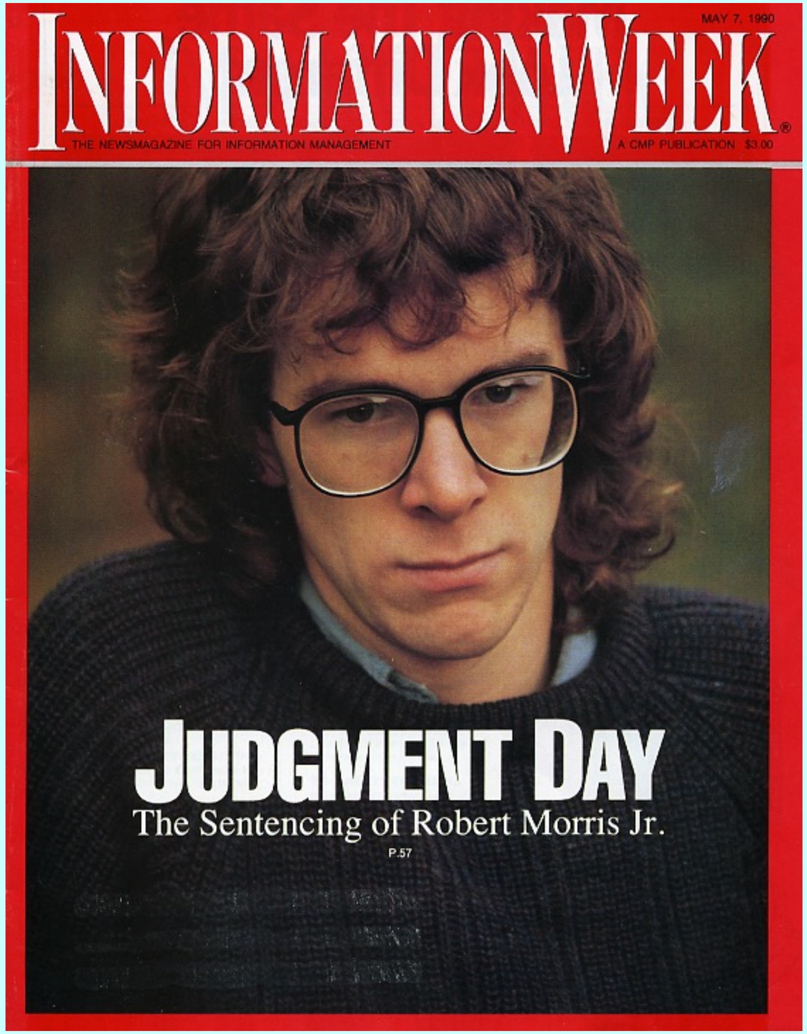
Nationally, the percentage of eligible Americans who are registered is estimated to be 70.9 percent, down 2.2 points from the 1984 level.

The group's study concluded that in many of the 30 states where final figures are available the decline was among

Robert Morris Jr.

Robert Morris Jr. is best known for creating the Morris Worm in 1988, considered the first computer worm on the Internet. In 1989, he was indicted for violating the Computer Fraud and Abuse Act. He was the first person to be indicted under this act. In December 1990, he was sentenced to three years of probation, 400 hours of community service, and a fine of $10,050 plus the costs of his supervision.

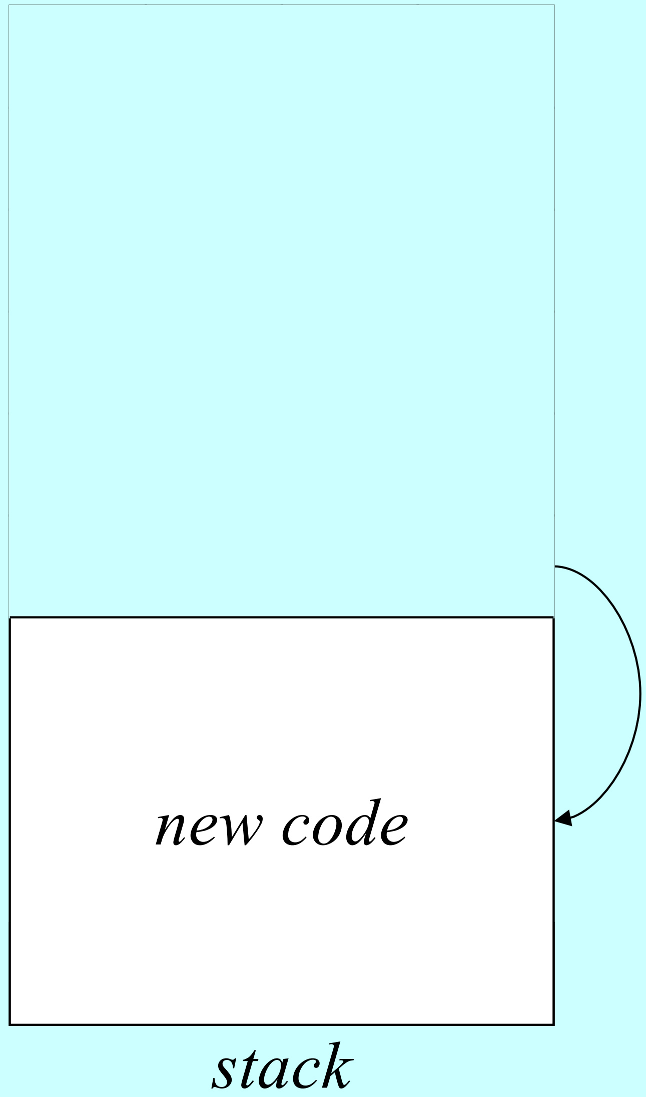
He is now a Professor at Massachusetts Institute of Technology, and an entrepreneur, e.g., Partner of Y Combinator.



How the Morris Worm Worked

If the user, however, enters a name string that overflows the buffer, the bytes in that name will overwrite the data on the stack.

Now when the function returns, it will jump into the code written as part of the name, thereby executing the worm's instructions.



*p++

```
#include <iostream>
#include <string>
using namespace std;

int main(void)
{
    int arr[] = {1, 2, 3, 4};
    int * p = arr;
    int a = *p++;
    // a = *(p++); i.e., a = *p; p = p + 1;
    int b = *++p;
    // b = *(++p); i.e., p = p + 1; b = *p;
    cout << "a = " << a << ", b = " << b << endl;
    return 0;
}

Output:
a = 1, b = 3
```

*p++

```
#include <iostream>
#include <string>
using namespace std;

int main(void)
{
    int arr[] = {1, 2, 3, 4};
    // arr is a non-modifiable lvalue
    int a = *arr++;
    // a = *arr; arr = arr + 1;
    int b = *++arr;
    // arr = arr + 1; b = *arr;
    cout << "a = " << a << ", b = " << b << endl;
    return 0;
}

Output:
a = ?, b = ?
```

Pointers and Arrays Example

```
int **ppi, *pi, i = 10;
pi = &i;
ppi = &pi;

&i: 0x6dfed4
i: 10
&pi: 0x6dfed8
pi: 0x6dfed4
*pi: 10
&ppi: 0x6dfedc
ppi: 0x6dfed8
*ppi: 0x6dfed4
**ppi: 10

double doubleArray[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};
double* doublePointer = doubleArray;
```

inters and Arrays Example

```
FE80
006DFE80
00000
0000000
6DFE88
&doubleArray[1]: 006DFE88
*doubleArray+1: 00000001
*(doubleArray+1): 00000002
doubleArray[1]: 00000002
doubleArray+9: 006DFEC8
&doubleArray[9]: 006DFEC8
*(doubleArray+9): 00000012
doubleArray[9]: 00000012
doubleArray+10: 006DFED0
&doubleArray[10]: 006DFED0
*(doubleArray+10): 00000000
doubleArray[10]: 00000000
doubleArray-1: 006DFE78
*doubleArray-1: FFFFFFFF
*(doubleArray-1): 00000000
&doubleArray: 006DFE80
&doubleArray+1: 006DFED0
*(&doubleArray+1): 006DFED0
&doubleArray-1: 006DFE30
*(&doubleArray-1): 006DFE30
```

Pointers and Arrays Example

```
doublePointer: 006DFE80
doublePointer+1: 006DFE88
&doublePointer: 006DFE7C
&doublePointer+1: 006DFE80
doublePointer[0]: 00000000
doublePointer[1]: 00000002
doublePointer[9]: 00000012
doublePointer[10]: 00000000
&doublePointer[0]: 006DFE80
&doublePointer[1]: 006DFE88
&doublePointer[9]: 006DFEC8
&doublePointer[10]: 006DFED0
*doublePointer: 00000000
*doublePointer+1: 00000001
*(doublePointer+1): 00000002
*doublePointer++: 00000000
*++doublePointer: 00000004
```

Pointers and Arrays Example

```
char charArray[] = "acegikmoqs";
char* charPointer = "acegikmoqs";

charArray: 006DFE71
charArray+1: 006DFE72
&charArray: 006DFE71
&charArray+1: 006DFE7C
charArray[0]: 00000061
charArray[1]: 00000063
charArray[9]: 00000073
charArray[10]: 00000000
&charArray[0]: 006DFE71
&charArray[1]: 006DFE72
&charArray[9]: 006DFE7A
&charArray[10]: 006DFE7B
*charArray: 00000061
*charArray+1: 00000062
*(charArray+1): 00000063
```

Pointers and Arrays Example

```
charPointer: 004BD40A
charPointer+1: 004BD40B
&charPointer: 006DFE6C
&charPointer+1: 006DFE70
charPointer[0]: 00000061
charPointer[1]: 00000063
charPointer[9]: 00000073
charPointer[10]: 00000000
&charPointer[0]: 004BD40A
&charPointer[1]: 004BD40B
&charPointer[9]: 004BD413
&charPointer[10]: 004BD414
*charPointer: 00000061
*charPointer+1: 00000062
*(charPointer+1): 00000063
*charPointer++: 00000061
*++charPointer: 00000065

charPointer: 004BD40C
charPointer+1: 004BD40D
doubleArray[10]: 2.22045E-313
```

Reference vs Pointer - Declare

```cpp
int x { 3 };

// declaration & initialization
int& xRef { x };

// modification
xRef = 10;
```

```cpp
int x { 3 };

// declaration
int& xRef;  // not compiled

// initialization
xRef = &x;  // not compiled

// modification
xRef = 10;
```

```cpp
int x { 3 };

// declaration
int* xPtr { &x };

// modification
*xPtr = 10;
```

```cpp
int x { 3 };

// declaration
int* xPtr { nullptr };
xPtr = &x;

// modification
*xPtr = 10;
```

Reference vs Pointer - Modify

```
int x { 3 };
int y { 4 };

auto & xRef = x;
auto & yRef = y;

xRef = &y; // not compiled
```

```
int x { 3 };
int y { 4 };

auto & xRef = x;
auto & yRef = y;

xRef = yRef; // x = y
```

```
int x { 3 };
int y { 4 };

// declaration
int* xPtr { nullptr };
xPtr = &x;
xPtr = &y;

// modification
*xPtr = 10; // y = 10
```

```
int x { 3 };
int y { 4 };

auto * xPtr = &x;
auto * yPtr = &y;

xPtr = yPtr;

*xPtr = 10;  // y = 10;
```

Reference vs Pointer - const

```
int & ref {3}; // not compiled
```

```
const int & ref { 3 };

ref = 4;      // not compiled;
```

```
int x { 3 };

auto * const xPtr = &x; // const ptr

*xPtr = 10;
```

```
int x { 3 }, y { 4 };

auto * yPtr = &y;
auto * const xPtr = &x; // const ptr

xPtr = yPtr;

*xPtr = 10;
```

Reference vs Pointer - const

Reference variable is immutable, const by default

```
int x { 3 };

// const ptr to const int
auto const * const xPtr = &x;

*xPtr = 10;
```

Reference vs Pointer - to

```cpp
int x { 3 };

auto * xPtr = &x;

// reference to pointer
// int * & rPtr = xPtr
auto & rPtr = xPtr;

*rPtr = 10; // x = 10
```

```cpp
int x { 3 };

auto & xRef = x;

// Pointer to reference
auto * xPtr = &xRef;

*xPtr = 10; // x = 10
```

```cpp
int x { 3 };

auto & xRef = x;

auto & & yRef = xRef;
```

```cpp
int x { 3 };

auto & xRef = x;

auto & * xPtr = xRef;
```

The End