C++ Program Design -- Arrays, Strings, Pointers, and References

Junjie Cao @ DLUT

Summer 2016

http://jjcao.github.io/cPlusPlus

Array

Arrays

- use a struct to aggregate many different data types into one identifier
- An **array** is an aggregate data type that lets us access many variables of the same type through a single identifier.

```
// allocate 30 integer variables (each with a different name)
int testScoreStudent1;
int testScoreStudent2;
int testScoreStudent3;
// ...
int testScoreStudent30;


int testScore[30]; // allocate 30 integer variables in a fixed
array
```

Array elements and subscripting

- **subscript operator ([])**
- For an array of length N, the array elements are numbered 0 through N-1! This is called the array’s **range**.

```
int prime[5]; // hold the first 5 prime numbers
    prime[0] = 2;
    prime[1] = 3;
    prime[2] = 5;
    prime[3] = 7;
    prime[4] = 11;
```

Array data types

- Arrays can be made from any data type.
    - Arrays can also be made from structs.

```
struct Rectangle
{
    int length;
    int width;
};
Rectangle rects[5]; // declare an array of 5 Rectangle
```

Array subscripts

- must always be an integral type (char, short, int, long, long long, etc… -- and strangely enough, bool)

```
 // using a literal (constant) index:
array[1] = 7; // ok

// using an enum (constant) index
enum Animals{
    ANIMAL_CAT = 2
};
array[ANIMAL_CAT] = 4; // ok

// using a variable (non-constant) index:
short index = 3;
array[index] = 7; // ok
```

Fixed array declarations

- When declaring a fixed array, the size of the array (between the square brackets) must be a compile-time constant.
- // using a literal constant
- int array[5]; // Ok
-
- // using a macro symbolic constant
- #define ARRAY_SIZE 5
- int array[ARRAY_SIZE]; // Syntactically okay, but don’t do this
-
- // using a symbolic constant
- const int arraySize = 5;
- int array[arraySize]; // Ok

array declarations

```
// using an enumerator
enum ArrayElements{    MAX_ARRAY_SIZE = 5};
int array[MAX_ARRAY_SIZE]; // Ok

// using a non-const variable
int size;
std::cin >> size;
int array[size]; // Not ok -- size is not a compile-time constant!

// using a runtime const variable
int temp = 5;
const int size = temp;
int array[size]; // Not ok -- size is a runtime constant, not a compi
le-time constant!
```

Initializing fixed arrays

- One way to initialize an array is to do it element by element

- **initializer list:** `int prime[5] = { 2, 3, 5, 7, 11 };`
- if there are less initializers in the list than the array can hold, the remaining elements are initialized to 0

```
int array[5] = { 7, 4, 5 }; // only initialize first 3 elements

// Initialize all elements to 0
int array[5] = { };
```

- In C++11, the uniform initialization syntax can be used instead:
- `int prime[5] { 2, 3, 5, 7, 11 };`

Omitted size

- The following two lines are equivalent:

```
int array[5] = { 0, 1, 2, 3, 4 }; // explicitly define size of the array
int array[] = { 0, 1, 2, 3, 4 }; // let initializer list set size of the array
```

Passing arrays to functions

```
void passValue(int value) {// value is a copy of the argument
    value = 99; }// so changing it here won't change the value of the argument
void passArray(int prime[5]) {// prime is the actual array
    prime[0] = 11; }// so changing it here will change the original argument!

int main(){
    int value = 1;     passValue(value);
    std::cout << "after passValue: " << value << "\n";

    int prime[5] = { 2, 3, 5, 7, 11 };     passArray(prime);
    std::cout << "after passArray: " << prime[0] << "\n";
    return 0;
}
```

```
// even though prime is the actual array, within this function i
t should be treated as a constant
void passArray(const int prime[5])
{
    // so each of these lines will cause a compile error!
    prime[0] = 11;
    prime[1] = 7;
    prime[2] = 5;
    prime[3] = 3;
    prime[4] = 2;
}
```

sizeof and arrays

```
void printSize(int array[]){
    std::cout << sizeof(array) << '\n'; // prints the size of a pointer,
not the size of the array!
}

int main(){
    int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
    std::cout << sizeof(array) << '\n'; // will print the size of the a
rray
    printSize(array);

    return 0;
}
```

this printed:

```
32
4
```

Indexing an array out of range

```
int main(){
    int prime[5]; // hold the first 5 prime numbers
    prime[5] = 13;

    return 0;}
```

- 13 will be inserted into memory where the 6th element would have bee n had it existed
- This could overwrite the value of another variable
- or cause your program to crash.
- *Rule: When using arrays, ensure that your indices are valid for the rang e of your array!*.

Quiz

- 1) Declare an array to hold the high temperature (to the nearest tenth of a degree) for each day of a year (assume 365 days in a year). Initialize the array with a value of 0.0 for each day.

- 2) Set up an enum with the names of the following animals: chicken, dog, cat, elephant, duck, and snake. Put the enum in a namespace. Define an array with an element for each of these animals, and use an initializer list to initialize each element to hold the number of legs that animal has.
- Write a main function that prints the number of legs an elephant has, using the enumerator.

```
namespace Animals{
     enum Animals {
          CHICKEN,          DOG,          CAT,          ELEPHANT,          DUCK,
          SNAKE,          MAX_ANIMALS
     };
}
int main(){
     int legs[Animals::MAX_ANIMALS] = { 2, 4, 4, 4, 2, 0 };
     std::cout << "An elephant has " << legs[Animals::ELEPHANT] << " legs.\n";
     return 0;
}
```

Why not use enum class?

Loops and arrays

```
int scores[] = { 84, 92, 76, 81, 56 };
const int numStudents = sizeof(scores) / sizeof(scores[0]);
int totalScore = 0;

// use a loop to calculate totalScore
for (int student = 0; student < numStudents; ++student)
    totalScore += scores[student];

double averageScore = static_cast<double>(totalScore) / numStude
nts;
```

Arrays and off-by-one errors

```
int scores[] = { 84, 92, 76, 81, 56 };
const int numStudents = sizeof(scores) / sizeof(scores[0]);

int maxScore = 0; // keep track of our largest score
for (int student = 0; student <= numStudents; ++student)
    if (scores[student] > maxScore)
        maxScore = scores[student];

std::cout << "The best score was " << maxScore << '\n';
```

Arrays and off-by-one errors

```
int scores[] = { 84, 92, 76, 81, 56 };
if (scores[5] > maxScore)
            maxScore = scores[5];
```

- But scores[5] is undefined!
- This can cause all sorts of issues, with the most likely being that scores[5] results in a garbage value. In this case, the probable result is that maxScore will be wrong.
- However, imagine what would happen if we inadvertently assigned a value to array[5]!
- We might overwrite another variable (or part of it), or perhaps corrupt something -- these types of bugs can be very hard to track down!

Quiz 1

- Print the following array to the screen using a loop:

```
const int arrayLength(9);
int array[arrayLength] = { 4, 6, 7, 3, 8, 2, 1, 9, 5 };
```

Sorting an array using selection sort

A case for sorting

How sorting works

- Sorting is generally performed by repeatedly comparing pairs of array elements, and swapping them if they meet some predefined criteria

```
#include <algorithm> // for std::swap, use <utility> instead if C++11

#include <iostream>

int main()

{

    int x = 2;

    int y = 4;

   std::cout << "Before swap: x = " << x << ", y = " << y << '\n';

    std::swap(x, y); // swap the values of x and y

    std::cout << "After swap:  x = " << x << ", y = " << y << '\n';

}
```

Implement & Optimize the following algorithms

- Selection sort
- Insertion sort
- Bubble sort

Multidimensional Arrays

- An array of arrays is called a **multidimensional array**.
- `int array[3][5]; // a 3-element array of 5-element arrays`
- row-major order: [3] row, [5] column
- Layout

```
[0][0]  [0][1]  [0][2]  [0][3]  [0][4] // row 0
[1][0]  [1][1]  [1][2]  [1][3]  [1][4] // row 1
[2][0]  [2][1]  [2][2]  [2][3]  [2][4] // row 2
```

- array[2][3] = 7;

Initializing two-dimensional arrays

```
int array[3][5] ={
{ 1, 2, 3, 4, 5 }, // row 0
{ 6, 7, 8, 9, 10 }, // row 1
{ 11, 12, 13, 14, 15 } // row 2
};
```

- replace missing initializers with 0

```
int array[3][5] ={
{ 1, 2  }, // row 0 = 1, 2, 0, 0, 0
{ 6, 7, 8 }, // row 1 = 6, 7, 8, 0, 0
{ 11, 12, 13, 14 } // row 2 = 11, 12, 13, 14, 0
};
```

can still be initialized to 0

```
int array[3][5] = { 0 };
```

Initializing two-dimensional arrays

- Right

```
int array[][5] ={
{ 1, 2, 3, 4, 5 },
{ 6, 7, 8, 9, 10 },
{ 11, 12, 13, 14, 15 }
};
```

- Wrong

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

Accessing elements in a two-dimensional array

```
for (int row = 0; row < numRows; ++row) // step through the rows in the array
    for (int col = 0; col < numCols; ++col)//step through each element in the row
        std::cout << array[row][col];
```

- **Multidimensional arrays larger than two dimensions**

```
int array[5][4][3];
std::cout << array[3][1][2];
```

String in C++

- String: a collection of sequential characters, such as “Hello, world!”
- C++
  - std::string
  - C-style strings: an array of characters that uses a null terminator.
    - A **null terminator** is a special character (‘\0’, ascii code 0) used to indicate the end of the string.

C-style strings

- To define a C-style string, simply declare a char array and initialize it with a string literal:

```
int main()
{
    char mystring[] = "string";
    std::cout << mystring << " has " << sizeof(mystring) << " ch
aracters.\n";
    for (int index = 0; index < sizeof(mystring); ++index)
        std::cout << static_cast<int>(mystring[index]) << " ";

    return 0;
}
```

```
string has 7 characters.
115 116 114 105 110 103 0
```

C-style strings follow all the same rules as arrays

- can initialize it upon creation, but can not assign values to it using the assignment operator after that!

```
char mystring[] = "string"; // ok
mystring = "rope"; // not ok!

int main(){
    char mystring[] = "string";
    mystring[1] = 'p';
    std::cout << mystring;

    return 0;
}
```

Print a c-style string

- std::cout prints characters until it encounters the null terminator.
- If you accidentally overwrite the null terminator in a string (e.g. by assigning something to mystring[6]), std::cout will just keep printing everything in adjacent memory slots until it happens to hit a 0!

```
int main()
{
    char name[20] = "Alex"; // only use 5 characters (4 letters
+ null terminator)
    std::cout << "My name is: " << name << '\n';


    return 0;
}
```

C-style strings and std::cin

- don’t know in advance how long our string is going to be.

```
int main()
{
    char name[255]; // declare array large enough to hold 255 ch
aracters
    std::cout << "Enter your name: ";
    std::cin >> name;
    std::cout << "You entered: " << name << '\n';

    return 0;
}
```

The recommended way of reading strings using cin

```
#include <iostream>
int main()
{
    char name[255]; // declare array large enough to hold 255 characters
    std::cout << "Enter your name: ";
    std::cin.getline(name, 255);//read up to 254 characters into name (leavin
g room for the null terminator!)
    std::cout << "You entered: " << name << '\n';

    return 0;
}
```

Manipulating C-style strings 1

```
char source[] = ″Copy this!″;
char dest[4]; // note that the size of dest is only 4 chars!
strcpy(dest, source); // overflow!
cout << dest;
```

Use strncpy instead of strcpy

```
char source[] = ″Copy this!″;
char dest[50];
strncpy(dest, source, 49); // copy at most 49 characters (indice
s 0-48)
dest[49] = 0; // ensures the last character is a null terminator
cout << dest; // prints ″Copy this!″
```

strlen()

- strlen() function, which returns the length of the C-style string (without the null terminator).

- Other useful functions:
  strcat() -- Appends one string to another (dangerous)
  strncat() -- Appends one string to another (with buffer length check)
  strcmp() -- Compare two strings (returns 0 if equal)
  strncmp() -- Compare two strings up to a specific number of characters (returns 0 if equal)

```
int main(){
    // Ask the user to enter a string
    char buffer[255];
    std::cout << "Enter a string: ";
    std::cin.getline(buffer, 255);

    int spacesFound = 0;
    // Loop through all of the characters the user entered
    for (int index = 0; index < strlen(buffer); ++index)    {
        // If the current character is a space, count it
        if (buffer[index] == ' ')    spacesFound++;
    }

    std::cout << "You typed " << spacesFound << " spaces!\n";
    return 0;}
```

- *Rule: Use std::string instead of C-style string*

Pointers

What is a variable?

- a name for a piece of memory that holds a value
- address-of operator (&) allows us to see what memory address is assigned to a variable

```
int main()
{
    int x = 5;
    std::cout << x << '\n'; // print the value of variable x
    std::cout << &x << '\n'; // print the memory address of variable x

    return 0;
}
```

```
the above program printed:
5
0027FEA0
```

The dereference operator (*)

- address-of operator (&)
- dereference operator (*) allows us to get the value at a particular address:

```
int main() {
    int x = 5;
    std::cout << x << '\n'; // print the value of variable x
    std::cout << &x << '\n'; // print the memory address of variable x
    std::cout << *&x << '\n'; // print the value at the memory address of var
iable x

    return 0;
}
```

Pointers

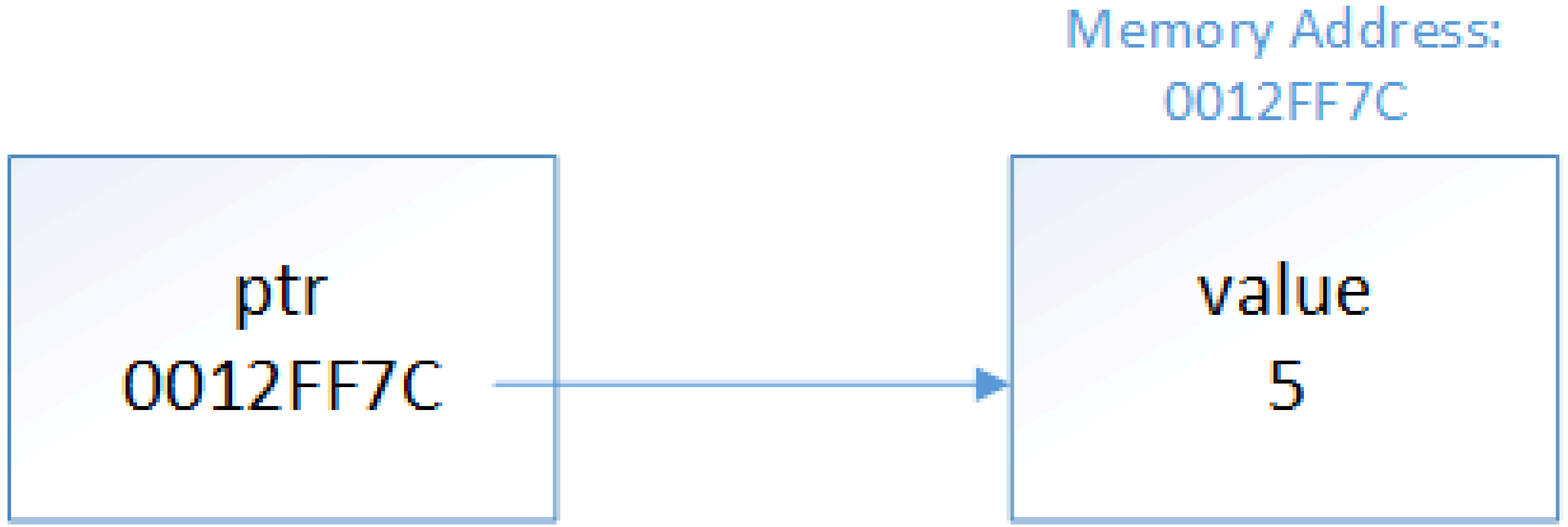
- Pointer variables are declared just like normal variable, only with an ast erisk between the data type and the variable name:
- `int *iPtr; // a pointer to an integer value`
- `double *dPtr; // a pointer to a double value`
-

- `int* iPtr2; // also valid syntax (acceptable, but not favored)`
- `int * iPtr3; // also valid syntax (but don't do this)`
- `int* iPtr6, iPtr7; // iPtr6 is a pointer to an int, but iPtr7 is just a plain int!`
-
- `int *iPtr4, *iPtr5; // declare two pointers to integer variables`
- For this reason, when declaring a variable, we recommend putting the asterisk next to the variable name.

- *Best practice: When declaring a function, put the asterisk of a pointer return value next to the type.*
- `int* doSomething();`

Assigning a value to a pointer

- Since pointers only hold addresses, when we assign a value to a pointer, that value has to be an address
- `int value = 5;`
- `int *ptr = &value; // initialize ptr with address of variable value`



```cpp
int main()
{
    int value = 5;
    int *ptr = &value; // initialize ptr with address of variable value

    std::cout << &value << '\n'; // print the address of variable value
    std::cout << ptr << '\n'; // print the address that ptr is holding

    return 0;
}
```

this printed:

```
0012FF7C
0012FF7C
```

The type of the pointer has to match the type of the variable being pointed to

- int iValue = 5;
- double dValue = 7.0;
-
- int *iPtr = &iValue; // ok
- double *dPtr = &dValue; // ok
- iPtr = &dValue; // wrong -- int pointer cannot point to the add ress of a double variable
- dPtr = &iValue; // wrong -- double pointer cannot point to the address of an int variable

not legal

- `int *ptr = 5; // not okay, treated as assigning an integer literal`

- `double *dPtr = 0012FF7C; // not okay, treated as assigning an integer literal`

The address-of operator returns a pointer

```
#include <typeinfo>
int main()
{
int x(4);
std::cout << typeid(&x).name();

return 0;
}
```

On Visual Studio 2013, this printed:

```
int *
```

Dereferencing pointers

```
int value = 5;
std::cout << &value; // prints address of value
std::cout << value; // prints contents of value

int *ptr = &value; // ptr points to value
std::cout << ptr; // prints address held in ptr, which is &value
std::cout << *ptr; // dereference ptr (get the value that ptr is
pointing to)
```

a pointer value can be reassigned to another value

```
int value1 = 5;
int value2 = 7;

int *ptr;

ptr = &value1; // ptr points to value1
std::cout << *ptr; // prints 5

ptr = &value2; // ptr now points to value2
std::cout << *ptr; // prints 7
```

Change value through pointer

- `int value = 5;`
- `int *ptr = &value; // ptr points to value`
-
- `*ptr = 7; // *ptr is the same as value, which is assigned 7`
- `std::cout << value; // prints 7`

A warning about dereferencing invalid pointers

- Pointers in C++ are inherently **unsafe**, and improper pointer usage is one of the best ways to crash your application.

- When a pointer is dereferenced, the application attempts to go to the memory location that is stored in the pointer and retrieve the contents of memory.
- For security reasons, modern operating systems sandbox applications to prevent them from improperly interacting with other applications, and to protect the stability of the operating system itself.
- If an application tries to **access a memory location not allocated to it by the operating system, the operating system may shut down** the application.

- The following program illustrates this, and will probably crash when you run it (go ahead, try it, you won’t harm your machine):

```
void foo(int *&p) { }


int main() {
    int *p; // Create an uninitialized pointer (that points to garbage)
    foo(p); // Trick compiler into thinking we’re going to assign this a valid value


    std::cout << *p; // Dereference the garbage pointer


    return 0;
}
```

The size of pointers

a pointer on a 32-bit machine is 32 bits (**4 bytes**). With a 64-bit executable, a p ointer would be 64 bits (8 bytes). Note that this is true regardless of what is bei ng pointed to:

```
char *chPtr; // chars are 1 byte
int *iPtr; // ints are usually 4 bytes
struct Something{
    int nX, nY, nZ;
};
Something *somethingPtr; // Something is probably 12 bytes

std::cout << sizeof(chPtr) << '\n'; // prints 4
std::cout << sizeof(iPtr) << '\n'; // prints 4
std::cout << sizeof(somethingPtr) << '\n'; // prints 4
```

What good are pointers?

- At this point, pointers may seem a little silly, academic, or obtuse. Why use a pointer if we can just use the original variable?
- useful in many different cases:
  - **Arrays** are implemented using pointers.
  - the only way you can **dynamically allocate memory** in C++. the most common use case for pointers.
  - **pass a large amount of data** to a function in a way that doesn’t involve copying the data, which is inefficient
  - achieve **polymorphism** when dealing with inheritance
  - have one struct/class point at another struct/class, to form a chain.
    - useful in some more **advanced data structures**, such as linked lists and trees.

Conclusion

- Pointers are variables that hold a memory address.
- They can be dereferenced using the dereference operator (*) to retrieve the value at the address they are holding.
- Dereferencing a garbage pointer may crash your application.

Quiz 1

```
short value = 7; // &value = 0012FF60
short otherValue = 3; // &otherValue = 0
012FF54


short *ptr = &value;


std::cout << &value << '\n';
std::cout << value << '\n';
std::cout << ptr << '\n';
std::cout << *ptr << '\n';
std::cout << '\n';
```

Quiz 2

```
short value = 7; // &value = 0012FF60
short otherValue = 3; // &otherValue = 0012FF54
short *ptr = &value;
*ptr = 9;

std::cout << &value << '\n';
std::cout << value << '\n';
std::cout << ptr << '\n';
std::cout << *ptr  << '\n';
std::cout << '\n';
```

Quiz 3

```
short value = 7; // &value = 0012FF60
short otherValue = 3; // &otherValue = 0012FF54
short *ptr = &value;
*ptr = 9;

ptr = &otherValue;

std::cout << &otherValue << '\n';
std::cout << otherValue << '\n';
std::cout << ptr << '\n';
std::cout << *ptr << '\n';
std::cout << '\n';

std::cout << sizeof(ptr) << '\n';
std::cout << sizeof(*ptr) << '\n';
```

Null pointers

- Just like normal variables, pointers are not initialized when they are instantiated.
    - Unless a value is assigned, a pointer will point to some garbage address by default.

```
double *ptr(0);
if (ptr)
    cout << ″ptr is pointing to a double value.″;
else
    cout << ″ptr is a null pointer.″;
```

- *Best practice: Initialize your pointers to a null value if you're not giving them another value*.

The NULL macro & nullptr in C++11

- `int *ptr(NULL); // assign address 0 to ptr`
- NULL is a marco (#define NULL 0) => avoid using it

- *Best practice: With C++11, use keyword **nullptr** to initialize your pointers to a null value*.
- `int *ptr = nullptr; // note: ptr is still an integer pointer, just set to a null value (0)`

Pointers and arrays

Similarities between pointers and fixed arrays

- We know what the values of array[0], array[1], … are 9, 7, …. But what value does array itself have?
- The variable array contains the address of the first element of the array, **as if it were a pointer**!

```
int main(){
    int array[5] = { 9, 7, 5, 3, 1 };
    // print the value of the array variable
    std::cout << "The array has address: " << array << '\n';
    // print address of the array elements
    std::cout << "Element 0 has address: " << &array[0] << '\n';

    return 0;}
```

```
The array has address: 0042FD5C
Element 0 has address: 0042FD5C
```

Differences between pointers and fixed arrays

- an array and a pointer to the array are not identical!
- different type information: int[5] vs. int *
- A fixed array knows how long it is. A pointer to the array does not.

```
int main(){
     int array[5] = { 9, 7, 5, 3, 1 };
     std::cout << sizeof(array) << '\n'; // will print sizeof(int) * arra
y length
     int *ptr = array;
     std::cout << sizeof(ptr) << '\n'; // will print the size of a pointer

return 0;}
```

Passing fixed arrays to functions

- copying large arrays can be very expensive, passing pointer instead

```
void printSize(int *array){// array is treated as a pointer here
     std::cout << sizeof(array) << '\n'; // prints the size of a pointer, n
ot the size of the array!
}

int main(){
    int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
    std::cout << sizeof(array) << '\n'; // will print sizeof(int) * ar
ray length
    printSize(array);


    return 0;}
```

implicitly convertion

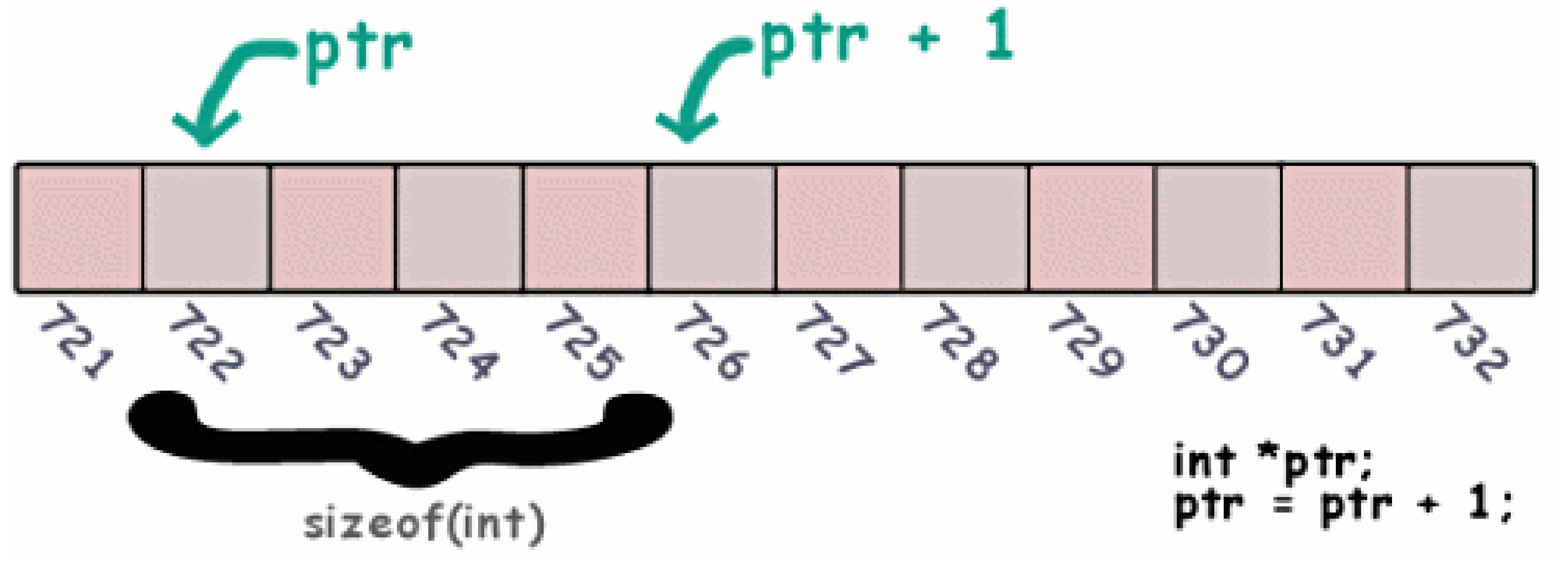
- C++ implicitly converts parameters using the array syntax ([]) to the poi nter syntax (*) => the following two are identical:
- `void printSize(int array[]);`
- `void printSize(int *array);`

An intro to pass by address

```
void changeArray(int *ptr){
    *ptr = 5; // so changing an array element changes the _actual_ array
}
int main(){
    int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
    std::cout << "Element 0 has value: " << array[0] << '\n';

    changeArray(array);

    std::cout << "Element 0 has value: " << array[0] << '\n';

    return 0;}
```

Pointer arithmetic

- C++ allows you to perform integer addition or subtraction operations on pointers.
- Scaling
  - ptr + 1 does not return the memory address after ptr, but the memory address of the next object of the type that ptr points to.



```
int *ptr;
ptr = ptr + 1;
```

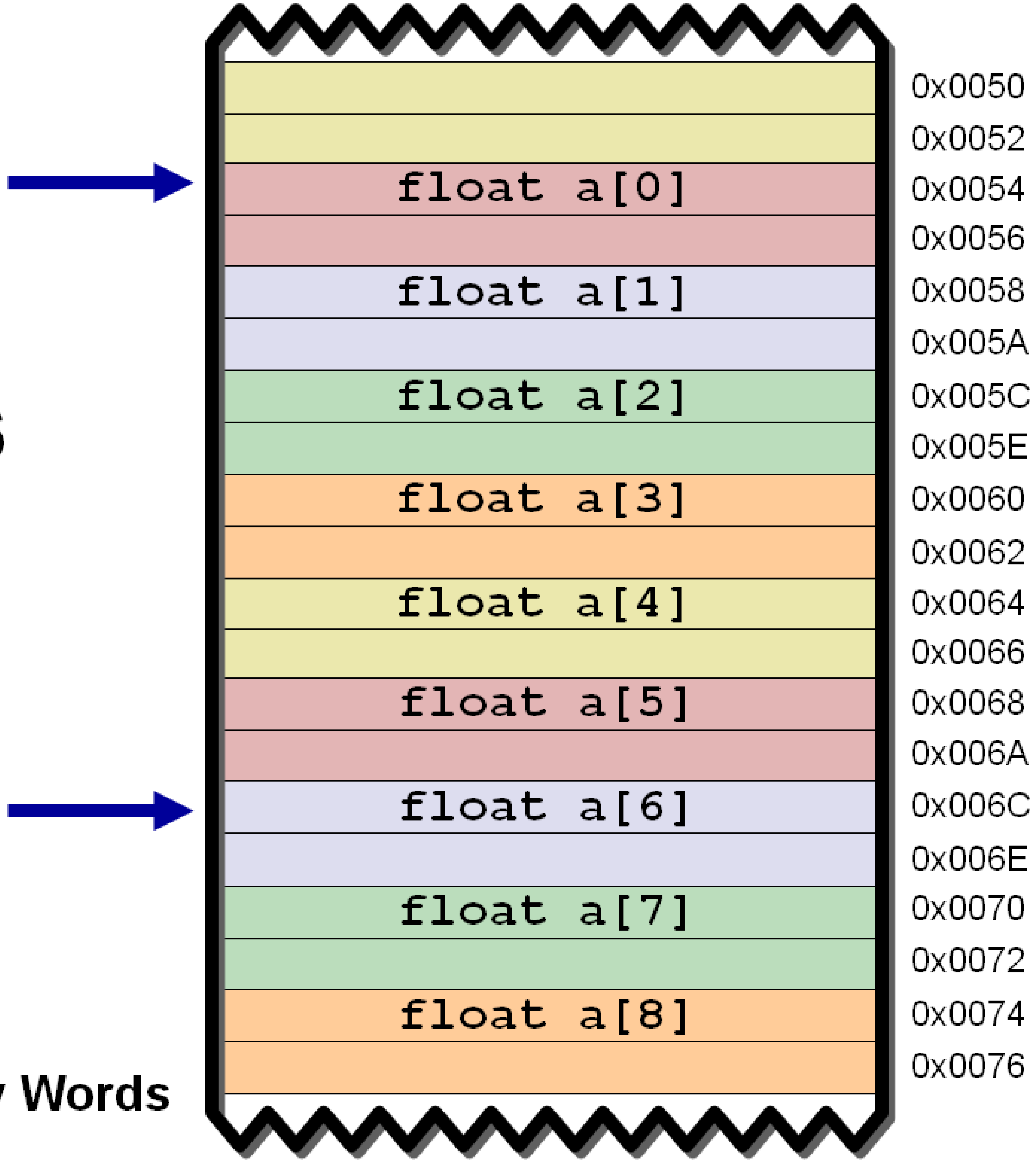
Arrays are laid out sequentially in memory

```
float *ptr;

          ptr = &a;
```

Adding 6 to `ptr` moves it 6 `float` array elements ahead (24 bytes ahead)

```
          ptr += 6;
```

16-bit Data Memory Words



Pointer arithmetic, arrays, and the magic behind indexing

```
int main(){
    int array [5] = { 9, 7, 5, 3, 1 };

    std::cout << &array[1] << '\n'; // memory address of array element 1
    std::cout << array+1 << '\n'; // memory address of array pointer + 1

    std::cout << array[1] << '\n'; // prints 7
    std::cout << *(array+1) << '\n'; // prints 7 (note the parenthesis re
quired here)

  return 0;}
```

```
0017FB80
0017FB80
7
7
```

Using a pointer to iterate through an array

```
const int arraySize = 7;
char name[arraySize] = "Mollie"; int numVowels(0);
for (char *ptr = name; ptr < name + arraySize; ++ptr){
    switch (*ptr)    {
        case 'A':    case 'a':        case 'E':    case 'e':
        case 'I':    case 'i':        case 'O':    case 'o':
        case 'U':    case 'u':
            numVowels++;
    }
}
cout << name << " has " << numVowels << " vowels.\n";
```

```
Mollie has 3 vowels
```

C-style string symbolic constants

- Fixed array case:
- `char myName[] = "Alex";`
- `std::cout << myName;`
- string symbolic constants using pointers
- `const char *myName = "Alex";`
- `std::cout << myName;`
- Difference?
  - free to alter the contents of the array
  - Usually compiler places the string “Alex\0” into read-only memory somewhere
- Multiple string literals with the same content may point to the same location.
- *Rule: Feel free to use C-style string symbolic constants if you need read-only strings in your program, but always make them const!*

std::cout and char pointers

```
int main(){
    int nArray[5] = { 9, 7, 5, 3, 1 };
    char cArray[] = "Hello!";
    const char *name = "Alex";

    std::cout << nArray << '\n'; // nArray will decay to type int*
    std::cout << cArray << '\n'; // cArray will decay to type char*
    std::cout << name << '\n'; // name is already type char*

    return 0;}
```

```
003AF738
Hello!
Alex
```

```
int main(){
    char c = ’Q’;
    std::cout << &c;
    return 0;
}
```

- the programmer is intending to print the address of variable c.
- However, &c has type char*, so std::cout tries to print this as a string!
- On the author’s machine, this printed:
  - Q╠╠╠╠╨╡4;¿■A

- **So test is important**

dynamic memory allocation

The need for dynamic memory allocation

- C++ supports three basic types of memory allocation

  - **Static memory allocation** happens for **static and global** variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.

  - **Automatic memory allocation** happens for **function parameters** and **local variables**. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited, as many times as necessary.

  - **Dynamic memory allocation**

static and automatic allocation

- Both static and automatic allocation have two things in common:
  - The size of the variable / array must be known at compile time.
  - Memory allocation and deallocation happens automatically (when the variable is instantiated / destroyed).

- If we have to declare the size of everything at compile time, the best we can do is try to make a guess the maximum size of variables we'll need and hope that's enough:
- `char name[25]; // let's hope their name is less than 25 chars!`
- `Record record[500]; // let's hope there are less than 500 records!`
- `Monster monster[40]; // 40 monsters maximum`
- `Polygon rendering[30000]; // this 3d rendering better not have more than 30,000 polygons!`

```
char name[25]; // let's hope their name is less than 25 chars!
Monster monster[40]; // 40 monsters maximum
```

- wasted memory

- most normal variables (including fixed arrays) are allocated in a portion of memory called the **stack**.
  - The amount of stack memory for a program is generally quite small
  - VC defaults the stack size to 1MB.

  - If you exceed this number, stack overflow will result, and the operating system will probably close down the program.

Dynamic memory allocation

- int *ptr = new int; // dynamically allocate an integer and assign the address to ptr so we can access it later
- *ptr = 7; // assign value of 7 to allocated memory

- int *ptr1 = new int (5); // use direct initialization
- int *ptr2 = new int { 6 }; // use uniform initialization

- // assume ptr has previously been allocated with operator new
- delete ptr; // return the memory pointed to by ptr to the opera ting system
- ptr = 0; // set ptr to be a null pointer (use nullptr instead o f 0 in C++11)

Dangling pointers

- `delete ptr;`

- The delete operator does not *actually* delete anything.
- It simply returns the memory being pointed to back to the operating system.
- The operating system is then free to reassign that memory to another application (or to this application again later).

- Pointers that are pointing to deallocated memory are called **dangling pointer**.

Dangling pointers

```
int main(){
    int *ptr = new int; // dynamically allocate an integer
    *ptr = 7; // put a value in that memory location

    delete ptr; // return the memory to the operating system.  ptr
is now a dangling pointer.
    std::cout << *ptr; // Dereferencing a dangling pointer will
cause undefined behavior
    delete ptr; // trying to deallocate the memory again will al
so lead to undefined behavior.

    return 0;}
```

Dangling pointers

```
int main(){
    int *ptr = new int; // dynamically allocate an integer
    int *otherPtr = ptr; // otherPtr is now pointed at that same 
memory location
    delete ptr; // return the memory to the operating system.  p
tr and otherPtr are now dangling pointers.
    ptr = 0; // ptr is now a nullptr
    // however, otherPtr is still a dangling pointer!
    return 0;}
```

Rule: To avoid dangling pointers, after deleting memory, set all pointers pointing to the deleted memory to 0 (or nullptr in C++11).

Operator new can fail

- By default, if new fails, a *bad_alloc* exception is thrown.
- If this exception isn’t properly handled, the program will simply terminate (crash) with an unhandled exception error.

```
int *value = new (std::nothrow) int; // ask for an integer’s worth o
f memory
if (!value) // handle case where new returned null
{
    std::cout << ”Could not allocate memory”;
    exit(1);
}
```

Null pointers and dynamic memory allocation

- Null pointers (pointers set to address 0 or nullptr) are particularly useful when dealing with dynamic memory allocation.

```
// If ptr isn't already allocated, allocate it
if (!ptr)
    ptr = new int;
```

- Deleting a null pointer has no effect:

```
if (ptr)
    delete ptr;
```

Instead, you can just write:

```
delete ptr;
ptr = 0;
```

Memory leaks

- Dynamically allocated memory effectively has no scope. That is, it stays allocated until it is explicitly deallocated or until the program ends

```
void doSomething(){
    int *ptr = new int;
}
```

- `ptr has no chance to be deleted forever!`
    - ptr is the only variable holding the address
    - ptr will go out of scope.
- This is called a **memory leak**.

Memory leaks

- Memory leaks eat up free memory while the program is running, making less memory available not only to this program, but to other programs as well.

- Programs with severe memory leak problems can eat all the available memory, causing the entire machine to run slowly or even crash.

- `int value = 5;`
- `int *ptr = new int; // allocate memory`
- `ptr = &value; // old address lost, memory leak results`

- `int *ptr = new int;`
- `ptr = new int; // old address lost, memory leak results`

Dynamically allocating arrays

```
std::cout << "Enter a positive integer: ";
int size;    std::cin >> size;

int *array = new int[size]; // use array new.  Note that size does not need to
be constant!

std::cout << "I allocated an array of size " << size << '\n';

array[0] = 5; // set element 0 to value 5

delete[] array; // use array delete to deallocate array
array = 0; // use nullptr instead of 0 in C++11
```

Dynamic arrays are almost identical to fixed arrays

- Array: compiler know its size
- Dynamic array: compiler does not remember its size

Initializing dynamically allocated arrays

- initialize a dynamically allocated array to 0, is simple:
  - `int *array = new int[size]();`

- Prior to C++11, there’s no easy way to initialize it to a non-zero value
  - `int *array = new int[size](5);` **//error C3074: an array cannot be initialized with a parenthesized initializer**

- starting with C++11

```
int fixedArray[5] = { 9, 7, 5, 3, 1 }; // initialize a fixed array in C++03
int *array = new int[5] { 9, 7, 5, 3, 1 }; // initialize a dynamic array in C++11
```

Quiz

- Write a program that:
  * Asks the user how many names they wish to enter.
  * Asks the user to enter each name.
  * Calls a function to sort the names (modify the selection sort code from lesson **6.4 -- Sorting an array using selection sort**)
  * Prints the sorted list of names.

- Hint: Use a dynamic array of std::string to hold the names.
  Hint: std::string supports comparing strings via the comparison operators < and >

Your output should match this:

- How many names would you like to enter? 5
- Enter name #1: Jason
- Enter name #2: Mark
- Enter name #3: Alex
- Enter name #4: Chris
- Enter name #5: John

- Here is your sorted list:
- Name #1: Alex
- Name #2: Chris
- Name #3: Jason
- Name #4: John
- Name #5: Mark

Pointers and const

Pointers and const

```
const int value = 5; // value is const
int *ptr = &value; // compile error: cannot convert const int* to int*
*ptr = 6; // change value to 6
```

- **pointer to a const value**

```
const int value = 5;
const int *ptr = &value; // this is okay, ptr is pointing to a ”
const int”
*ptr = 6; // not allowed, we can’t change a const value
```

pointer to a const value

- Thus, the following is okay:

```
int value = 5;
const int *ptr = &value; // ptr points to a "const int"
value = 6; // the value is non-const when accessed through a non
-const identifier, *ptr is 6 now
```

- But the following is not:

```
int value = 5;
const int *ptr = &value; // ptr points to a "const int"
*ptr = 6; // ptr treats its value as const, so changing the valu
e through ptr is not legal
```

pointer to a const value

```
int value1 = 5;
const int *ptr = &value1; // ptr points to a const int

int value2 = 6;
ptr = &value2; // okay, ptr now points at some other const int
```

Const pointers

- A **const pointer** is a pointer whose value can not be changed after initialization
- A **pointer to a const value** is a (non-const) pointer that points to a constant value.

```
int value1 = 5;
int value2 = 6;

int * const ptr = &value1; // okay, the const pointer is initial
ized to the address of value1
ptr = &value2; // not okay, once initialized, a const pointer ca
n not be changed.
```

Const pointers

- A **const pointer** is a pointer whose value can not be changed after initialization

- A **pointer to a const value** is a (non-const) pointer that points to a constant value.

```
int value = 5;
int *const ptr = &value; // ptr will always point to value
*ptr = 6; // allowed, since ptr points to a non-const int
```

Const pointer to a const value

```
int value = 5;
const int *const ptr = &value;
```

Recapping

- To summarize, you only need to remember 4 rules, and they are pretty logical:
  - A non-const pointer can be redirected to point to other addresses.
  - A const pointer always points to the same address, and this address can not be changed.
  - A pointer to a non-const value can change the value it is pointing to. These can not point to a const value.
  - A pointer to a const value treats the value as const (even if it is not), and thus can not change the value it is pointing to.

- int value = 5;
- const int *ptr1 = &value; // ptr1 points to a "const int", so t
  his is a pointer to a const value.
- int *const ptr2 = &value; // ptr2 points to an "int", so this i
  s a const pointer to a non-const value.

Reference variables

Reference variables

- Normal variables, which hold values directly.

- Pointers, which hold the address of another value (or null) and can be dereferenced to retrieve the value at the address they point to.

- References are the third basic type of variable that C++ supports.

References

- A **reference** is a type of C++ variable that acts as an alias to another variable.

```
int value = 5; // normal integer
int &ref = value; // reference to variable value
value = 6; // value is now 6
ref = 7; // value is now 7

cout << value; // prints 7
++ref;
cout << value; // prints 8
```

Using the address-of operator on a reference

```
int value = 5; // normal integer
int &ref = value; // reference to variable value
value = 6; // value is now 6
ref = 7; // value is now 7

cout << &value; // prints 0012FF7C
cout << &ref; // prints 0012FF7C
```

References are implicitly const

- Reference to a constant variable
  - `const int x = 5;`
  - `int &ref = x; // invalid, non-const reference to const object`
  - `const int &ref = x; // OK`
- Const reference
  - `int value1 = 5;`
  - `int value2 = 6;`
  - `int &invalidRef; // invalid, needs to reference something`

  - `int &ref = value1; // okay, ref is now an alias for value1`
  - `ref = value2; // assigns 6 (the value of value2) to value1 -- does NOT change the reference!`

References as function parameters

```
// ref is a reference to the argument passed in, not a copy
void changeN(int &ref){ref = 6;}

int main(){
  int n = 5;
  std::cout << n << '\n';
  changeN(n); // note that this is a non-reference argument
  std::cout << n << '\n';

  return 0;
}
```

Rule: Pass non-pointer, non-fundamental data type variables by (const) reference.

References as shortcuts

```
struct Something
{
    int value1;
    float value2;
};
struct Other
{
    Something something;
    int otherValue;
};

Other other;
```

```
int &ref = other.something.value1;
// ref can now be used in place of oth
er.something.value1
```

The following two statements are thus identi
cal:

```
other.something.value1 = 5;
ref = 5;
```

References vs pointers

- *ptr and ref evaluate identically.

- Because **references** must be initialized to valid objects and can not be changed once set, references are generally much **safer** to use than pointers.

- However, they are also a bit more limited in functionality.

- If a given task can be solved with either a reference or a pointer, the reference should generally be preferred.

Member selection with pointers and references

```
struct Person
{
     int age;
     double weight;
};
Person person;

// Member selection using actual struct variable
person.age = 5;
```

- `Person &ref = person;`
- `ref.age = 5;`

```
ptr->age = 5;
```

For-each loops

For-each loops

- C++11 introduces a new type of loop called a **for-each** loop

```
for (element_declaration : array)
    statement;
```

```
int main()
{
    int fibonacci[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
    for (auto number : fibonacci) // type is auto, so number has its type deduced from the fibonacci array
        std::cout << number << ' ';

    return 0;
}
```

For-each loops

```
int array[5] = { 9, 7, 5, 3, 1 };
for (auto &element: array) // The ampersand makes element a reference to the actual array element, preventing a copy from being made
{
    std::cout << element << ’ ’;
}
```

Rule: Use references or const references for your element declaration in for-each loops for performance reasons.

For-each doesn’t work with pointers to an array

```
int sumArray(int array[]){
    int sum = 0;
    for (const auto &number : array) // compile error, the size of array isn't known
        sum += number;
    return sum;
}

int main()
{
     int array[5] = { 9, 7, 5, 3, 1 };
     std::cout << sumArray(array);
     return 0;
}
```

Quiz

- Declare a fixed array with the following names: Alex, Betty, Caroline, Dave, Emily, Fred, Greg, and Holly. Ask the user to enter a name. Use a for each loop to see if the name the user entered is in the array.

- Sample output:
  - Enter a name: Betty
  - Betty was found.
  - Enter a name: Megatron
  - Megatron was not found.

- Hint: Use std::string as your array type

Void pointers

Void pointers

- A void pointer can point to objects of any data type:

```
int nValue;float fValue;
struct Something{
    int n;    float f;
};

Something sValue;

void *ptr;
ptr = &nValue; // valid
ptr = &fValue; // valid
ptr = &sValue; // valid
```

Void pointers

- it cannot be dereferenced directly!

```
int value = 5;
void *voidPtr = &value;

//cout << *voidPtr << endl; // illegal: cannot dereference a void pointer

int *intPtr = static_cast<int*>(voidPtr); // however, if we cast our void pointer to an int pointer...

cout << *intPtr << endl; // then we can dereference it like normal
```

- If a void pointer doesn't know what it's pointing to, how do we know what to cast it to?
- Ultimately, that is up to you to keep track of.

```
void printValue(void *ptr, Type type){
    switch (type) {
        case INT:
            std::cout << *static_cast<int*>(ptr) << '\n';
            break;
        case CSTRING:
            std::cout << static_cast<char*>(ptr) << '\n';
            break;
    }
}
```

Avoid using void *

- avoid using void pointers unless absolutely necessary
  - as they effectively allow you to avoid type checking.
  - This allows you to do things that make no sense, and the compiler w on’t complain about it.

```
int nValue = 5;
    printValue(&nValue, CSTRING);
```

- who knows what the result would actually be!

Quiz

- What’s the difference between a void pointer and a null pointer?

- A void pointer is a pointer that can point to any type of object, but does not know what type of object it points to.
- A void pointer must be explicitly cast into another type of pointer to be dereferenced.

- A null pointer is a pointer that does not point to an address.
- A void pointer can be a null pointer.

Pointers to pointers

Pointers to pointers

```
int value = 5;
int *ptr = &value;
std::cout << *ptr; // dereference pointer to int to get int value

int **ptrptr = &ptr;
std::cout << **ptrptr; // first dereference to get pointer to in
t, second dereference to get int value
```

- `int value = 5;`
- `int **ptrptr = &&value; // not valid`

Arrays of pointers

- Pointers to pointers have a few uses. The most common use is to dynamically allocate an array of pointers:

-
  ```
  int **array = new int*[10]; // allocate an array of 10 int pointers
  ```

Two-dimensional dynamically allocated arrays

- Another common use

```
int **array = new int*[10]; // allocate an array of 10 int point
ers — these are our rows
for (int count = 0; count < 10; ++count)
    array[count] = new int[5]; // these are our columns

array[9][4] = 3; // This is the same as (array[9])[4] = 3;
```

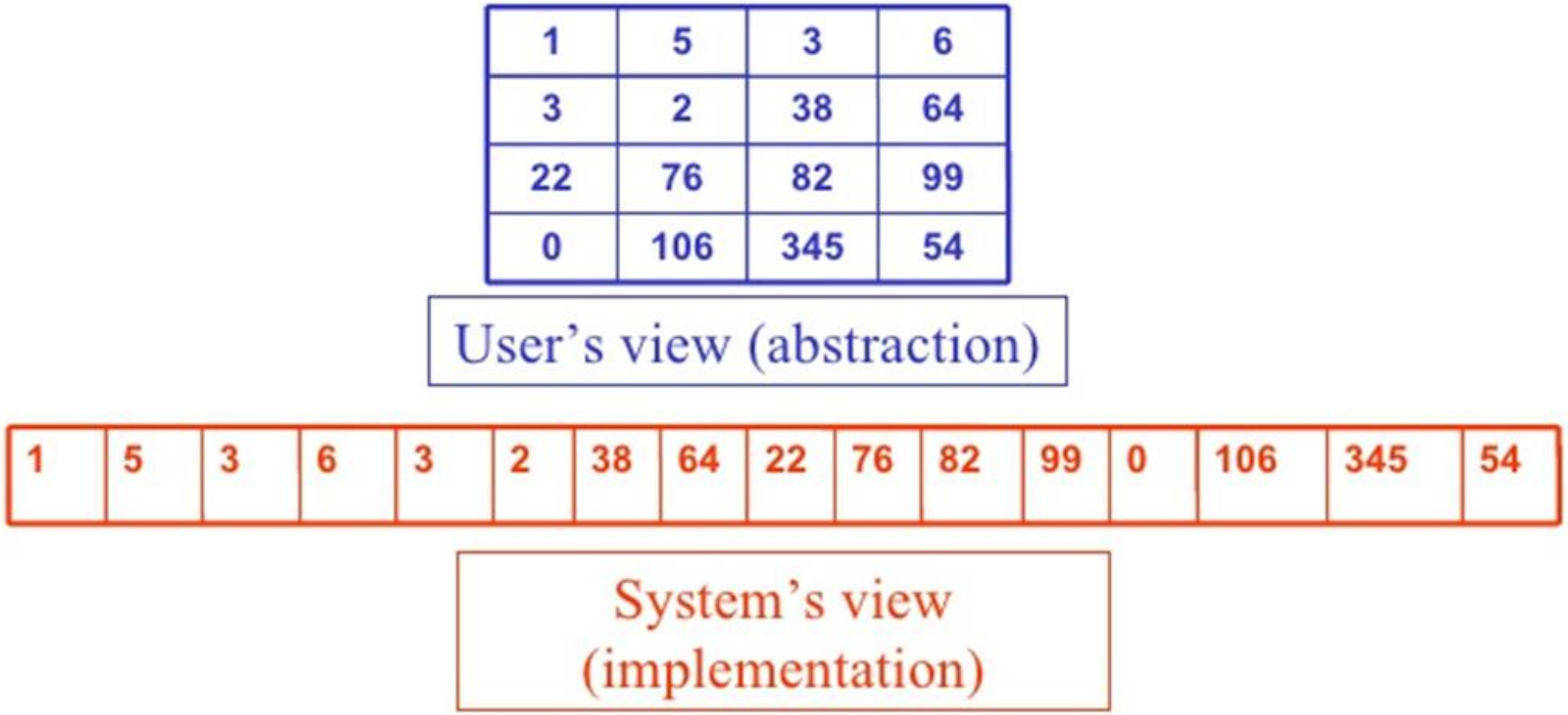
Deallocating

Deallocating a dynamically allocated two-dimensional array using this me thod requires a loop as well:

```
for (int count = 0; count < 10; ++count)
    delete[] array[count];
delete[] array; // this needs to be done last
```

allocating and deallocating two-dimensional arrays is complex and easy to mess up

- easier to “flatten” a two-dimensional array (of size x by y) into a one-dimensional array of size x * y:



1	5	3	6
3	2	38	64
22	76	82	99
0	106	345	54

1	5	3	6	3	2	38	64	22	76	82	99	0	106	345	54

Offset of a[i][j]?

two-dimensional arrays

```
int *array = new int[50]; // a 10x5 array flattened into a single array
int getSingleIndex(int row, int col, int numberOfColumnsInArray)
{
    return (row * numberOfColumnsInArray) + col;
}

// set array[9,4] to 3 using our flattened array
array[getSingleIndex(9, 4, 5)] = 3;
```

Conclusion

- We recommend avoiding using pointers to pointers unless no other options are available,
- because they’re complicated to use and potentially dangerous.

std::array

An introduction to std::array in C++11

- `#include <array>`
- `std::array<int, 5> myarray; // declare an integer array with length 3`
- `myarray = { 0, 1, 2, 3, 4 }; // okay`
- `myarray = { 9, 8, 7 }; // okay, elements 3 and 4 are set to zero!`
- `myarray = { 0, 1, 2, 3, 4, 5 }; // not allowed, too many elements in initializer list!`

- `std::array<int, 5> myarray2 { 9, 7, 5, 3, 1 }; // uniform initialization`

at() has bounds checking, but () hasn’t

- `std::array<int, 5> myarray { 9, 7, 5, 3, 1 };`

- `myarray.at(1) = 6; // array element 1 valid, sets array element 1 to value 6`
- `myarray.at(9) = 10; // array element 9 is invalid, will throw error`

- `myarray[9] = 6; // bad things will probably happen, but who knows, no exception thrown`

Size and sorting

- Because std::array doesn’t decay to a pointer when passed to a function, the size() function will work even if you call it from within a function:

```
void printSize(const std::array<double, 5> &myarray){
    std::cout << ”size: ” << myarray.size();
}


int main(){
    std::array<double, 5> myarray { 9.0, 7.2, 5.4, 3.6, 1.8 };

    printSize(myarray);

    return 0;}
```

```cpp
#include <array>
#include <algorithm> // for std::sort

int main(){
    std::array<int, 5> myarray { 7, 3, 1, 9, 5 };
    std::sort(myarray.begin(), myarray.end()); // sort the array forwar
ds
//    std::sort(myarray.rbegin(), myarray.rend()); // sort the array backwards

    for (const auto &element : myarray)
        std::cout << element << ' ';

    return 0;}
```

Summary

- std::array is a great replacement for build-in fixed arrays.

- It’s efficient, in that it doesn’t use any more memory than built-in fixed arrays.

- using std::array over built-in fixed arrays for any non-trivial use.

std::vector

std::vector

```
#include <vector>

// no need to specify size at initialization
std::vector<int> array;
// use initializer list to initialize array
std::vector<int> array2 = { 9, 7, 5, 3, 1 };
// use uniform initialization to initialize array (C++11 onward)
std::vector<int> array3 { 9, 7, 5, 3, 1 };

array[2] = 2;
array.at(3) = 3;
```

Self-cleanup prevents memory leaks

- As of C++11,
- `array = { 0, 1, 2, 3, 4 }; // okay, array size is now 5`
- `array = { 9, 8, 7 }; // okay, array size is now 3`

- `std::vector<int> array { 9, 7, 5, 3, 1 };`
- `std::cout << "The size is: " << array.size() << '\n';`

Resizing an array

```
std::vector<int> array { 0, 1, 2 };
array.resize(5); // set size to 5

std::cout << "The size is: " << array.size() << '\n';

for (auto const &element: array)
    std::cout << element << ' ';
```

```
The size is: 5
0 1 2 0 0
```

- when we resized the array, the existing element values were preserved!,
- new elements are initialized to the default value for the type (which is 0 for integers)

Conclusion

- std::vector handle their own memory management (which helps prevent memory leaks),
- remember their size,
- and can be easily resized,

- **using std::vector in almost all cases where dynamic arrays are needed.**

comprehensive quiz

- Arrays, C-style strings
- Be careful not to index an array out of the array’s range.

- Pointers * and Dereference operator (*)
- new, delete, new [], delete [], & memory leak
- A null pointer is a pointer that is not pointing at anything.
- Reference variable & and Address-of operator (&)

- Pointer to const and const pointer
- Represent 2D matrix as 1D array

- std::array, std::vector

Quiz time

- What’s wrong with each of these snippets, and how would you fix it?

```
int main()
{
    int array[5] { 0, 1, 2, 3 };
    for (int count = 0; count <= 5; ++count)
        std::cout << array[count] << ” ”;

    return 0;
}
```

Quiz time

- What’s wrong with each of these snippets, and how would you fix it?

```
int main() {
    int x = 5;     int y = 7;

    const int *ptr = &x;
    std::cout << *ptr;
    *ptr = 6;
    std::cout << *ptr;
    ptr = &y;
    std::cout << *ptr;

    return 0;}
```

Quiz time

- What’s wrong with each of these snippets, and how would you fix it?

```
void printArray(int array[]){
     for (const int &element : array)
     std::cout << element << ' ';
}


int main(){
     int array[] { 9, 7, 5, 3, 1 };
     printArray(array);

return 0;}
```

Quiz time

- What’s wrong with each of these snippets, and how would you fix it?

```
int* allocateArray(const int length)
{
    int temp[length];
    return temp;
}
```

Quiz time

- What’s wrong with each of these snippets, and how would you fix it?

```
int main()
{
    double d(5.5);
    int *ptr = &d;
    std::cout << ptr;

    return 0;
}
```