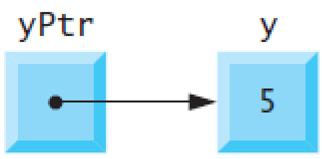
C++ Basics
Pointers

Pointer constants

There are four ways to pass a pointer to a function:

- a *nonconstant pointer to nonconstant data*,
- a *nonconstant pointer to constant data*,
- a *constant pointer to nonconstant data,*
- a *constant pointer to constant data*.



nonconstant pointer to nonconstant data

The highest access is granted by a **nonconstant pointer to nonconstant data**:

- the *data can be modified* through the dereferenced pointer, and
- the *pointer can be modified* to point to other data.

```
int* countPtr;
```

nonconstant pointer to constant data

A **nonconstant pointer to constant data** is:

• a pointer that can be modified to point to *any* data of the appropriate type, but

• the data to which it points *cannot* be modified through that pointer.

const int* countPtr;

❖ Some programmers prefer to write this as **int const* countPtr**; to make it obvious that const applies to the int, not the pointer. They'd read this declaration from right to left as "countPtr is a pointer to a constant integer."

nonconstant pointer to constant data

```cpp
void f(const int*); // prototype

int main() {
   int y{0};

   f(&y); // f will attempt an illegal modification
}

// constant variable cannot be modified through xPtr
void f(const int* xPtr) {
   *xPtr = 100; // error: cannot modify a const object
}
```

GNU C++ compiler error message:

```
fig08_10.cpp: In function 'void f(const int*)':
fig08_10.cpp:17:12: error: assignment of read-only location '* xPtr'
```

constant pointer to nonconstant data

A **constant pointer to nonconstant data** is a pointer that:

- always points to the same memory location, and
- the data at that location *can* be modified through the pointer.

int* <u>const</u> countPtr;

constant pointer to nonconstant data

```
int main() {
   int x, y;

   // ptr is a constant pointer to an integer that can be modified
   // through ptr, but ptr always points to the same memory location.
   int* const ptr{&x}; // const pointer must be initialized

   *ptr = 7; // allowed: *ptr is not const
   ptr = &y; // error: ptr is const; cannot assign to it a new address
}
```

Microsoft Visual C++ compiler error message:

```
'ptr': you cannot assign to a variable that is const
```

constant pointer to constant data

The *minimum* access privilege is granted by a **constant pointer to constant data**:

- such a pointer *always* points to the *same* memory location, and
- the data at that location *cannot* be modified via the pointer.

<u>const</u> int* <u>const</u> countPtr;

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

  // nonconst pointer to nonconst data
  int* xPtr1 = &a;
  *xPtr1 = 10;
  xPtr1 = tmpPtr;
  xPtr1 = tmpPtr2;
}
```

Examples

```
int main() {
   int* tmpPtr = nullptr;
   const int* tmpPtr2 = nullptr;
   int a{7}; // initialize a with 7

   // nonconst pointer to nonconst data
   int* xPtr1 = &a;
   *xPtr1 = 10; // OK
   xPtr1 = tmpPtr; // OK
   xPtr1 = tmpPtr2; // Error
}
```

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

  // nonconst pointer to const data
  const int* xPtr2 = &a;
  *xPtr2 = 10;
  xPtr2 = tmpPtr;
  xPtr2 = tmpPtr2;
}
```

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

  // nonconst pointer to const data
  const int* xPtr2 = &a;
  *xPtr2 = 10; // Error
  xPtr2 = tmpPtr; // OK
  xPtr2 = tmpPtr2; // OK
}
```

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

// const pointer to nonconst data
  int* const xPtr3 = &a;
  *xPtr3 = 10;
  xPtr3 = tmpPtr;
  xPtr3 = tmpPtr2;
}
```

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

// const pointer to nonconst data
  int* const xPtr3 = &a;
  *xPtr3 = 10; // OK
  xPtr3 = tmpPtr; // Error
  xPtr3 = tmpPtr2; // Error
}
```

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

// const pointer to const data
  const int* const xPtr4 = &a;
  *xPtr4 = 10;
  xPtr4 = tmpPtr;
  xPtr4 = tmpPtr2;
  int y = *xPtr4;
  tmpPtr = xPtr4;
  tmpPtr2 = xPtr4;
}
```

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

// const pointer to const data
  const int* const xPtr4 = &a;
  *xPtr4 = 10; // Error
  xPtr4 = tmpPtr; // Error
  xPtr4 = tmpPtr2; // Error
  int y = *xPtr4; // OK
  tmpPtr = xPtr4; // Error
  tmpPtr2 = xPtr4; // OK
}
```

C-style (built-in) array

C-style array is declared as:

type **arrayName** [*arraySize*];

The compiler reserves the appropriate amount of **<u>continuous</u>** memory at **compile time** and allocates the array in **stack**. The *arraySize* must be an **integer constant** greater than zero.

For example, to tell the compiler to reserve 12 elements for integer array c, use the declaration

int c[12]; // c is an array of 12 integers

C-style (built-in) array

Memory can be reserved for several arrays with a single declaration:

```
int b[ 100 ], // b is an array of 100 integers
    x[ 27 ]; // x is an array of 27 integers
```

❖ *It is recommended to declare one array per declaration for readability, modifiability and ease of commenting:*

```
int b[ 100 ]; // b is an array of 100 integers
int  x[ 27 ]; // x is an array of 27 integers
```

➢ Arrays can be declared to contain values of any **non-reference** data type:

```
int a = 5;
int b = 6;
int c = 7;
int& refArr[3] {a, b, c}; // ERROR!
```

C-style (built-in) array

```cpp
// Fig. 7.4: fig07_04.cpp
// Initializing an array in a declaration.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

int main()
{
   // use initializer list to initialize array n
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output each array element's value
   for ( int i = 0; i < 10; i++ )
      cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;

   return 0; // indicates successful termination
} // end main
```

```
Element        Value
      0           32
      1           27
      2           64
      3           18
      4           95
      5           14
      6           90
      7           70
      8           60
      9           37
```

C-style array initialization

If there are fewer initializers than elements in the array, the remaining array elements are initialized to **zero**:

```
int n[ 10 ] = { 0 }; // initialize elements of array n to 0
```

This declaration will leave the array uninitialized:

```
int n[ 10 ]; // uninitialized array!
```

C-style array initialization

```
int arr[5];
for (int i = 0; i < 5; ++i) {
    cout << arr[i] << endl;
}
```

Result:

0
2
104
15036204
6487912

C-style array initialization

```
int arr[5] {0};
for (int i = 0; i < 5; ++i) {
    cout << arr[i] << endl;
}
```

<u>Result</u>:
0
0
0
0
0

C-style array initialization

```
int arr[5] {0, 1};
for (int i = 0; i < 5; ++i) {
    cout << arr[i] << endl;
}
```

Result:
0
1
0
0
0

- If you provide **fewer** initializers than the number of elements, the remaining elements are ***value initialized***—fundamental numeric types are set to **0**, bools are set to **false**, pointers are set to **nullptr** and class objects are **initialized by their default constructors.**

C-style array initialization

```
static int arr[5];
for (int i = 0; i < 5; ++i) {
    cout << arr[i] << endl;
}
```

Result:
0
0
0
0
0

- If a static array is not explicitly initialized, its elements are value initialized: **fundamental numeric types** are set to **0**, **bools** are set to **false**, **pointers** are set to **nullptr** and **class objects** are initialized by their **default constructors**.

C-style array initialization

- If the array size is <u>omitted</u> from a declaration with an initializer list, the compiler determines the number of elements in the array <u>by counting the number of elements in the initializer list</u>:

```
int n[] = { 1, 2, 3, 4, 5 }; // creates a five-element array.
```

- If the array size and an initializer list are specified in an array declaration, the number of initializers must be less than or equal to the array size:

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 }; // Error: 6 element
```

❖ Providing **more initializers** in an array initializer list than there are elements in the array is a <u>compilation</u> error.

❖ **Forgetting to initialize the elements** of an array whose elements should be initialized is a <u>logic</u> error.

C-style array initialization

The size of the array must be **a compile time constant**:

```
int x = 5;
int arr[x]; // Error

const int x = 5;
int arr[x]; // OK
```

- Defining the **size** of each array as a **constant** variable instead of a **literal constant** can make programs more **scalable**.

- Defining the size of an array as a constant variable instead of a literal constant makes programs **clearer**. This technique eliminates so-called **magic numbers**.

Passing arrays to functions

To pass an array argument to a function, specify the name of the array without any brackets. For example,

```
int hourlyTemperatures[ 24 ];
```

This function call passes array hourlyTemperatures and its size to function modifyArray:

```
modifyArray( hourlyTemperatures, 24 );
```

Passing arrays to functions

The function header will be

```
void modifyArray( int b[], int arraySize )
```

Note the strange appearance of the function prototype for modifyArray

```
void modifyArray( int [], int );
```

This prototype could have been written

```
void modifyArray( int anyArrayName[], int anyVariableName );
```

but C++ compilers <u>ignore</u> variable names in prototypes

- C++ passes arrays to functions **by reference** (we will see how it is done).

Passing arrays to functions

```
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

void modifyArray( int [], int ); // appears strange
void modifyElement( int );

int main()
{
  const int arraySize = 5; // size of array a
  int a[ arraySize ] = { 0, 1, 2, 3, 4 }; // initialize array a

  cout << "Effects of passing entire array by reference:"
    << "\n\nThe values of the original array are:\n";

  // output original array elements
  for ( int i = 0; i < arraySize; i++ )
    cout << setw( 3 ) << a[ i ];

  cout << endl;
```

Passing arrays to functions

```
   // pass array a to modifyArray by reference
   modifyArray( a, arraySize );
   cout << "The values of the modified array are:\n";

   // output modified array elements
   for ( int j = 0; j < arraySize; j++ )
      cout << setw( 3 ) << a[ j ];

   cout << "\n\n\nEffects of passing array element by value:"
      << "\n\na[3] before modifyElement: " << a[ 3 ] << endl;

   modifyElement( a[ 3 ] ); // pass array element a[ 3 ] by value
   cout << "a[3] after modifyElement: " << a[ 3 ] << endl;

   return 0; // indicates successful termination
} // end main
```

```
// in function modifyArray, "b" points to the original array "a" in memory
void modifyArray( int b[], int sizeOfArray )
{
   // multiply each array element by 2
   for ( int k = 0; k < sizeOfArray; k++ )
      b[ k ] *= 2;
} // end function modifyArray

// in function modifyElement, "e" is a local copy of
// array element a[ 3 ] passed from main
void modifyElement( int e )
{
   // multiply parameter by 2
   cout << "Value of element in modifyElement: " << ( e *= 2 ) << endl;
} // end function modifyElement
```

```
Effects of passing entire array by reference:

The values of the original array are:
   0   1   2   3   4
The values of the modified array are:
   0   2   4   6   8


Effects of passing array element by value:

a[3] before modifyElement: 6
Value of element in modifyElement: 12
a[3] after modifyElement: 6
```

Passing arrays to functions as const

```
#include <iostream>
using std::cout;
using std::endl;

void tryToModifyArray( const int [] ); // function prototype

int main()
{
   int a[] = { 10, 20, 30 };

   tryToModifyArray( a );
   cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';

   return 0; // indicates successful termination
} // end main

// In function tryToModifyArray, "b" cannot be used
// to modify the original array "a" in main.
void tryToModifyArray( const int b[] )
{
   b[ 0 ] /= 2; // error
   b[ 1 ] /= 2; // error
   b[ 2 ] /= 2; // error
} // end function tryToModifyArray
```

Passing arrays to functions as const

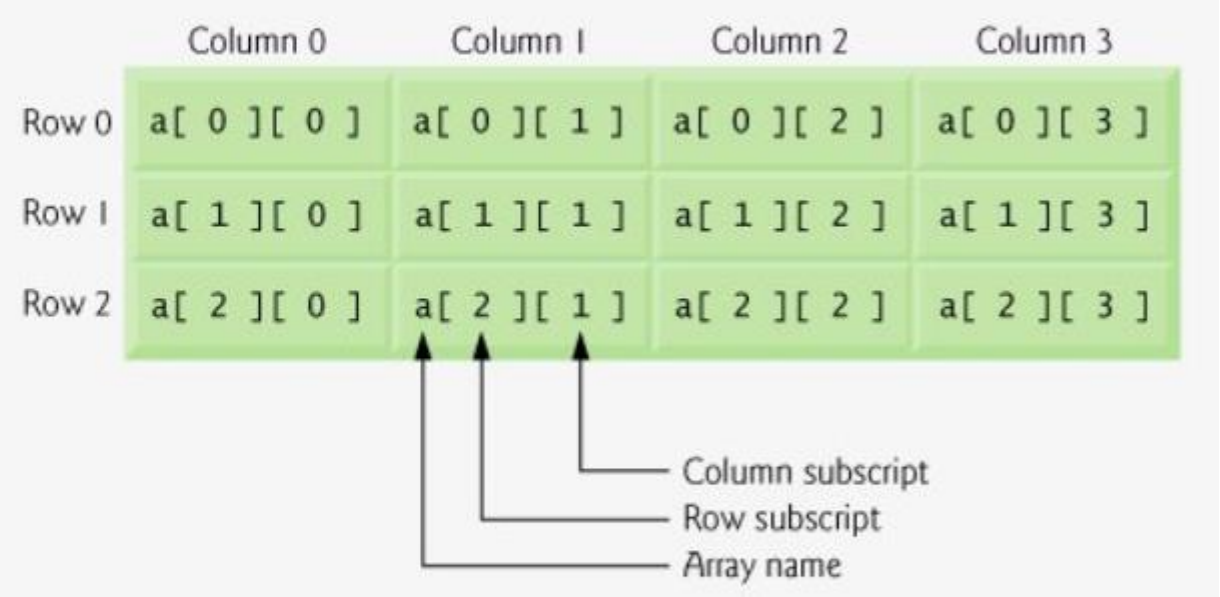
Borland C++ command-line compiler error message:

```
Error E2024 fig07_15.cpp 23: Cannot modify a const object
    in function tryToModifyArray(const int * const)
 Error E2024 fig07_15.cpp 24: Cannot modify a const object
    in function tryToModifyArray(const int * const)
 Error E2024 fig07_15.cpp 25: Cannot modify a const object
    in function tryToModifyArray(const int * const)
```

- When a function specifies an **array parameter** that is preceded by the **const** qualifier, **the elements of the array become constant in the function body**, and any attempt to modify an element of the array in the function body results in a compilation error.

Multidimensional arrays

Multidimensional arrays with two dimensions are often used to represent **tables of values** consisting of information arranged in rows and columns.



- Referencing a two-dimensional array element **a[x][y]** incorrectly as **a[x, y]** is an error. *Actually, a[x, y] is treated as a[y],* because C++ evaluates the expression x, y (containing a comma operator) simply as y (the last of the comma-separated expressions).

Multidimensional arrays

A multidimensional array can be initialized in its declaration much like a one-dimensional array: (the values are **grouped by row** in braces)

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0:

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
        // b[0][0] = 1;
        // b[0][1] = 0;
        // b[1][0] = 3;
        // b[1][1] = 4;
```

Multidimensional arrays

```
#include <iostream>
using std::cout;
using std::endl;

void printArray( const int [][ 3 ] ); // prototype

int main()
{
   int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
   int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
   int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

   cout << "Values in array1 by row are:" << endl;
   printArray( array1 );

   cout << "\nValues in array2 by row are:" << endl;
   printArray( array2 );

   cout << "\nValues in array3 by row are:" << endl;
   printArray( array3 );
   return 0; // indicates successful termination
} // end main
```

Multidimensional arrays

```
// output array with two rows and three columns
void printArray( const int a[][ 3 ] )
{
  // loop through array's rows
  for ( int i = 0; i < 2; i++ )
  {
    // loop through columns of current row
    for ( int j = 0; j < 3; j++ )
      cout << a[ i ][ j ] << ' ';

    cout << endl; // start new line of output
  } // end outer for
} // end function printArray
```

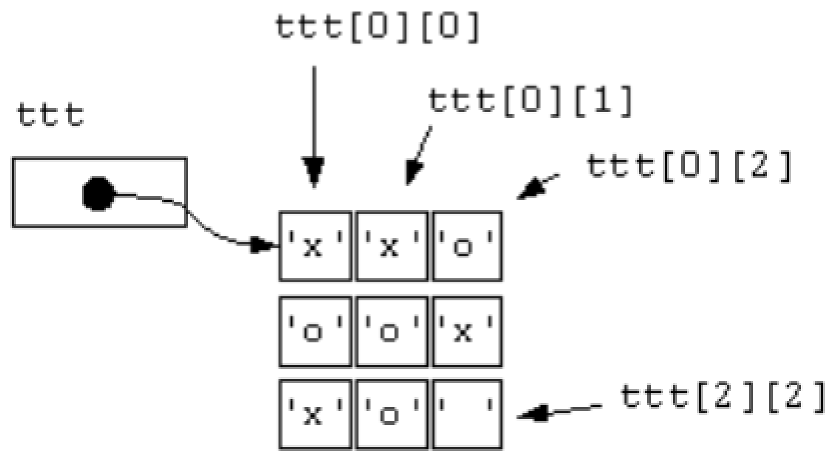
Result:

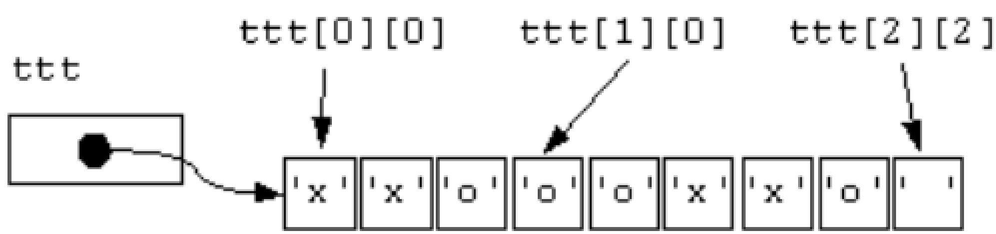
```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Multidimensional arrays

<u>Notice</u> that the function definition specifies the parameter **const int a[][3]**.

```
char ttt[3][3] = {{'x', 'x', 'o'},
                  {'o', 'o', 'x'},
                  {'x', 'o', ' '}
                 };
```





- **All array elements are stored <u>consecutively</u> in memory, regardless of the number of dimensions.**

Thus, when accessing a[1][2], the function knows to skip **row 0's three elements** in memory to **get to row 1**. Then, the function accesses **element 2 of that row**.

Character array

Character array can be initialized using a string literal.

```
char string1[] = "first";
```

- The size of array **string1** in the preceding declaration is determined by the compiler based on the **length** of the string.
- The string "first" contains **five** characters plus a **special string-termination character called the <u>null character</u>**.
- The character constant representation of the null character is **‘\0’.**

Character arrays also can be initialized with individual character constants in an initializer list:

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

Converting a string to uppercase

```
#include <iostream>
using std::cout;
using std::endl;

#include <cctype> // prototypes for islower and toupper
using std::islower;
using std::toupper;

void convertToUppercase( char * );

int main()
{
   char phrase[] = "characters and $32.98";

   cout << "The phrase before conversion is: " << phrase;
   convertToUppercase( phrase );
   cout << "\nThe phrase after conversion is: " << phrase << endl;
   return 0; // indicates successful termination
} // end main
```

Converting a string to uppercase

```
// convert string to uppercase letters
void convertToUppercase( char *sPtr )
{
  while ( *sPtr != '\0' ) // loop while current character is not '\0'
  {
    if ( islower( *sPtr ) ) // if character is lowercase,
      *sPtr = toupper( *sPtr ); // convert to uppercase

    sPtr++; // move sPtr to next character in string
  } // end while
 } // end function convertToUppercase
```

Result:

The phrase before conversion is: **characters and $32.98**

The phrase after conversion is: **CHARACTERS AND $32.98**

sizeof operator

The compile time unary operator sizeof determines the size in bytes of a built-in array or of any other data type, variable or constant during program compilation.

```
#include <iostream>
using std::cout;
using std::endl;

size_t getSize( double * ); // prototype

int main()
{
   double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system

   cout << "The number of bytes in the array is " << sizeof( array );

   cout << "\nThe number of bytes returned by getSize is "
      << getSize( array ) << endl;
   return 0; // indicates successful termination
} // end main
```

sizeof operator

```
// return size of ptr
size_t getSize( double *ptr )
{
   return sizeof( ptr );
} // end function getSize
```

<u>Result:</u>

The number of bytes in the array is 160
The number of bytes returned by getSize is 4

- Using the **sizeof** operator in a function to find *the size in bytes of a built-in array parameter* results in the *size in bytes of a pointer*, <u>not the size in bytes of the built-in array</u>.

sizeof operator

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

int main() {
   char c; // variable of type char
   short s; // variable of type short
   int i; // variable of type int
   long l; // variable of type long
   long long ll; // variable of type long long
   float f; // variable of type float
   double d; // variable of type double
   long double ld; // variable of type long double
   int array[20]; // built-in array of int
   int* ptr{array}; // variable of type int *
```

sizeof operator

```
    cout << "sizeof c = " << sizeof c
       << "\tsizeof(char) = " << sizeof(char)
       << "\nsizeof s = " << sizeof s
       << "\tsizeof(short) = " << sizeof(short)
       << "\nsizeof i = " << sizeof i
       << "\tsizeof(int) = " << sizeof(int)
       << "\nsizeof l = " << sizeof l
       << "\tsizeof(long) = " << sizeof(long)
       << "\nsizeof ll = " << sizeof ll
       << "\tsizeof(long long) = " << sizeof(long long)
       << "\nsizeof f = " << sizeof f
       << "\tsizeof(float) = " << sizeof(float)
       << "\nsizeof d = " << sizeof d
       << "\tsizeof(double) = " << sizeof(double)
       << "\nsizeof ld = " << sizeof ld
       << "\tsizeof(long double) = " << sizeof(long double)
       << "\nsizeof array = " << sizeof array
       << "\nsizeof ptr = " << sizeof ptr << endl;
}
```

```
sizeof c = 1    sizeof(char) = 1
sizeof s = 2    sizeof(short) = 2
sizeof i = 4    sizeof(int) = 4
sizeof l = 8    sizeof(long) = 8
sizeof ll = 8   sizeof(long long) = 8
sizeof f = 4    sizeof(float) = 4
sizeof d = 8    sizeof(double) = 8
sizeof ld = 16  sizeof(long double) = 16
sizeof array = 80
sizeof ptr = 8
```

Pointer arithmetic

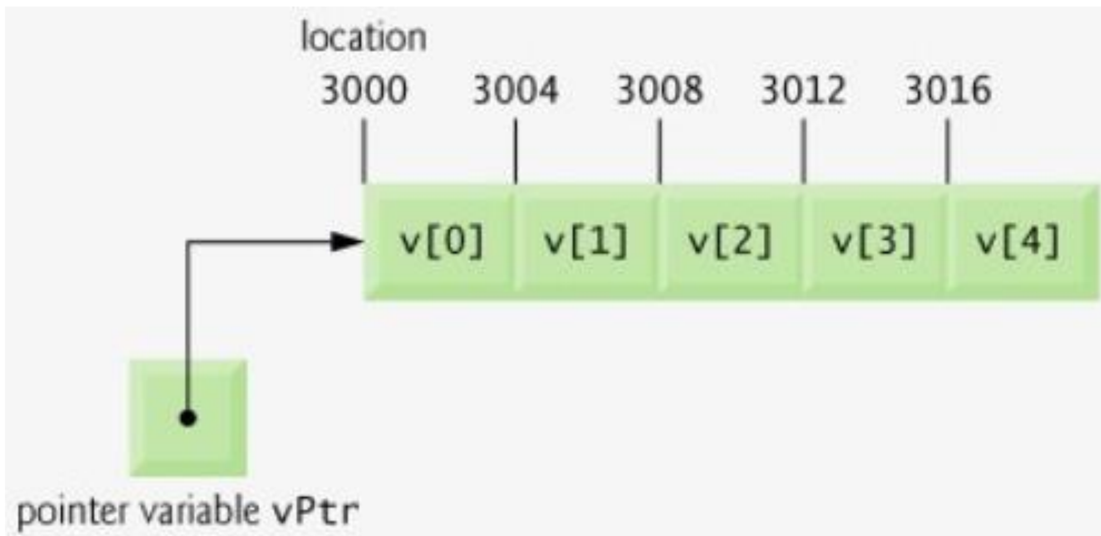
Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.

A pointer may be

- incremented (++) or decremented (--),
- an integer may be added to a pointer (+ or +=),
- an integer may be subtracted from a pointer (- or -=)
- or one pointer may be subtracted from another.

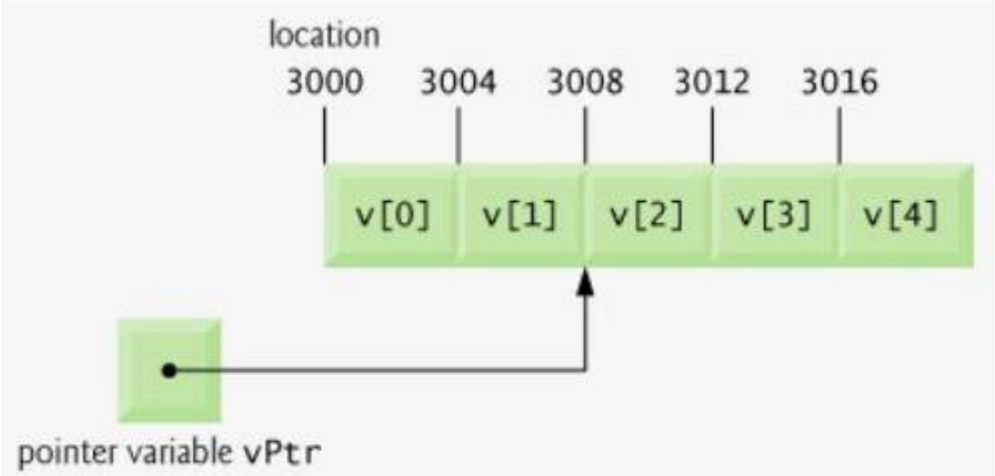
Pointer arithmetic

```
int v[5];
int *vPtr = v;  // OR   int *vPtr = &v[ 0 ];
```



An array name can be thought of as a <u>constant pointer.</u>

Pointer arithmetic



vPtr += 2; -> (3000 + 2 * sizeof(int))
vPtr -= 4; -> 3016 – 4*sizeof(int)
++vPtr; // OK
vPtr++; // OK
--vPtr; // OK
vPtr--; // OK
x = v2Ptr - vPtr; // (3008-3000)/sizeof(int)

Pointer arithmetic

- Using pointer arithmetic on a pointer that **does not refer to an array of values** is a logic error.

- Subtracting or comparing two pointers that **do not refer to elements of the same array** is a logic error.

- Using pointer arithmetic to *increment* or *decrement a pointer* such that the pointer refers to an element past the end of the array or before the beginning of the array is normally a logic error.

- *There's no bounds checking on pointer arithmetic.*

void* pointer

- A pointer can be assigned to another pointer if both pointers are of the ***same*** **type.**

- The **pointer to void** (i.e., **void***), is a generic pointer *capable of representing any pointer type*.

- Any pointer to a fundamental type or class type can be assigned to a pointer of type void* without casting.

- A pointer of type void* *cannot* be assigned directly to a pointer of another type - the pointer of type void* must first be ***cast*** to the proper pointer type

❖ *Assigning a pointer of one type to a pointer of another (other than void*) without using a cast is a compilation error.*

void* pointer

```
int main()
{
   int a = 10;
   char b = 'x';

   void* p = &a;  // void pointer holds address of int 'a'
   p = &b; // void pointer holds address of char 'b'
}
```

void* pointer

```
int main()
{
   int a = 10;
   void* ptr = &a;

   cout << *ptr;

   return 0;
}
```

Compiler Error: 'void*' is not a pointer-to-object type

void* pointer

```
int main()
{
   int a = 10;
   void* ptr = &a;

   cout << *(int *)ptr << endl;

   return 0;
}
```

Result: 10

void* pointer

- A void* pointer *cannot* be dereferenced.

- For example, the compiler "knows" that an int* points to four bytes of memory on a machine with four-byte integers. Dereferencing an int* creates an *lvalue* that is an alias for the int's four bytes in memory.

- A void*, however, simply contains a memory address for an *unknown* data type. **You cannot dereference a void*** because the compiler does *not* know the type of the data to which the pointer refers and thus not the number of bytes.

- *The allowed operations on void* pointers are:* ***comparing*** *void* pointers with other pointers,* ***casting*** *void* pointers to other pointer types and* ***assigning*** *addresses to void* pointers. All other operations on void* pointers are compilation errors.*

Pointer arithmetic

- Pointers can be **compared** using equality and relational operators. Comparisons using relational operators are meaningless unless **the pointers point to elements of the *same* built-in array.** Pointer comparisons compare the *addresses* stored in the pointers.

```
int v[5] = {1,2,3,4,5};

int* vptr1 = &v[0];
int* vptr2 = &v[4];

while (vptr1 < vptr2) { // prints all elements except the last one
    std::cout << *vptr1 << std::endl;
    vptr1++;
}
```

Result:
1
2
3
4

Relationship Between Pointers and Arrays

An <u>array name</u> can be thought of as a **constant pointer**. Pointers can be used to do any operation involving array *subscripting*.

```
int b[ 5 ]; // create 5-element int array b
int *bPtr; // create int pointer bPtr

bPtr = b; // assign address of array b to bPtr
bPtr = &b[ 0 ]; // also assigns address of array b to bPtr
```

Array element **b[3]** can alternatively be referenced with the pointer expression

*(bPtr + 3) or
*(b + 3)

The 3 in the preceding expression is the **offset** to the pointer.
The preceding notation is referred to as **pointer/offset notation**.

&b[3] => (bPtr + 3) or (b + 3)

Relationship Between Pointers and Arrays

Pointers can be subscripted exactly as arrays can. For example, the expression

bPtr[1]

refers to the array element **b[1].**
This expression uses **pointer/subscript notation**.

- Remember that an array name is a **constant pointer**; it always points to the beginning of the array. Thus, the expression

  b += 3

  causes a compilation error.

❖ For clarity, use array notation instead of pointer notation when manipulating arrays.

Relationship Between Pointers and Arrays

```
int main()
{
   int b[] = { 10, 20, 30, 40 }; // create 4-element array b
   int *bPtr = b; // set bPtr to point to array b

   // output array b using array subscript notation
   cout << "Array b printed with:\n\nArray subscript notation\n";

   for ( int i = 0; i < 4; i++ )
      cout << "b[" << i << "] = " << b[ i ] << '\n';

   // output array b using the array name and pointer/offset notation
   cout << "\nPointer/offset notation where "
      << "the pointer is the array name\n";

   for ( int offset1 = 0; offset1 < 4; offset1++ )
      cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';
```

Relationship Between Pointers and Arrays

```
   // output array b using bPtr and array subscript notation
   cout << "\nPointer subscript notation\n";

   for ( int j = 0; j < 4; j++ )
      cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';

   cout << "\nPointer/offset notation\n";

   // output array b using bPtr and pointer/offset notation
   for ( int offset2 = 0; offset2 < 4; offset2++ )
      cout << "*(bPtr + " << offset2 << ") = "
         << *( bPtr + offset2 ) << '\n';

   return 0; // indicates successful termination
} // end main
```

Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40

Bad example!

```
int a = 5;
int b = 6;
int c = 7;

int* p = &a;
p[2] = 5555; // undefined behavior

cout << a << " " << b << " " << c << endl;
```

Result: 5 6 **5555**