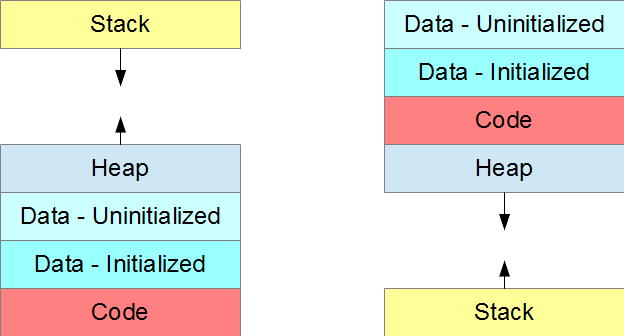
**Arrays and Pointers to Arrays**

An array may have one or more dimensions and may occupy data, stack or freestore (heap) memory.  An array type is a built-in type that represents a data structure of identical types stored contiguously in memory and provides an efficient layout for random access to its data values.  We refer to each data value in an array as an *element*.  The C++ language does not specify how the allocation of an array is implemented.  The name of an array is a pointer to the start of its region of memory.  A pointer can not only store the address of a single object but also the address of an array of objects of identical type.

  
  
Memory Regions Occupied by an Executing Instance of an Application

The size of an array in data and stack memory is determined at compile-time, while the size of an array in freestore memory is determined at run-time.  Sizes of arrays allocated in data or stack memory cannot be modified at run-time.

This chapter describes one-dimensional and multi-dimensional arrays in these different regions of memory.  This chapter also describes ragged arrays and introduces the syntax for pointing to arrays of objects.

One-Dimensional Arrays

Compile-Time Memory

The size of an array in the data or stack region of memory is known at compile-time.  The definition of a one-dimensional array in data or stack memory takes the form

|  |
| --- |
| *Type identifier*[ *c* ] *= initial*; |

*Type* is the type of every element in the array.  *c* is the number of elements in the array and is an integer constant or constant integer expression.  *= initial* is an optional, braces-enclosed, comma-separated list of initial values.  If this initialization list is present, *c* is optional.  If c exceeds the number of values in the list, the compiler initializes the uninitialized elements to 0.  If the initialization list is absent, *c* is required.

Run-Time Memory

The definition of a one-dimensional array allocated at run-time consists of three statements:

1. the definition of a pointer to hold the address of the array
2. the allocation of memory for the array
3. the deallocation of the allocated memory

The pointer definition takes the form

|  |
| --- |
| *Type*\* *identifier*; |

The allocation of dynamic memory uses the new[] operator:

|  |
| --- |
| new *Type*[ *c* ] |

The allocation reserves a contiguous region of memory for c elements and returns the address of the start of the region.  We store the address in the previously defined pointer

|  |
| --- |
| *identifier* = new *type*[ *c* ]; |

We deallocate the memory before our pointer's name goes out of scope:

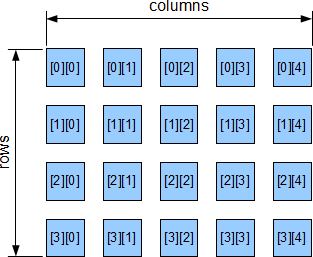
|  |
| --- |
| delete [] *identifier*; |

The following example shows the dynamic allocation and deallocation of a one-dimensional array

|  |  |
| --- | --- |
| // Dynamic Allocations of an Array  // dynamic.cpp  #include <iostream>  #include <iomanip>  int main () {  int c;  int \*a; // pointer to array a  std::cout << "Number of elements : ";  std::cin >> c;  a = new int[c]; // allocate memory  for (i = 0; i < c; i++)  a[i] = (i + 1) \* (i + 1);  for (i = 0; i < c; i++)  std::cout << std::setw(3) << i + 1  << std::setw(6) << a[i] << std::endl;  delete [] a; // deallocate memory  } | Number of elements : 5  1 1  2 4  3 9  4 16  5 25 |

Multi-Dimensional Arrays

Multi-dimensional arrays are arrays of two or more dimensions.  A two-dimensional array uses two independent indices to refer to its elements.  The indices of a two-dimensional array identify the row and column positions in a table of elements.  Indexing is zero-based.



The left index is the row index and the right index is the column index.

|  |
| --- |
| *identifier*[ *row* ][ *column* ] |

Row-Major Order

The compiler stores the elements of a two-dimensional array in *row-major order*: the first row, column-element by column-element, then the second row, column-element by column-element, then the third row, etc..  For example, it stores the elements of the array

|  |
| --- |
| int a[2][3]; |

as follows

|  |
| --- |
| a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2] |

We may interpret a two-dimensional array as an array of one-dimensional arrays

|  |
| --- |
| a[0][0] a[0][1] a[0][2] first row or 1-d array  a[1][0] a[1][1] a[1][2] second row or 1-d array |

Compile-Time Memory

The definition of a two-dimensional array determined at compile-time takes the form

|  |
| --- |
| *type identifier*[ *r* ][ *c* ] *= initial*; |

r is the number of rows in the array and c is the number of columns.  r and c are integer constants or constant integer expressions.  The total number of elements in the array is r \* c.  = initial is an optional initialization list.  = initial is a braces-enclosed, comma-separated list of initial values for the elements.  Each subset of initial values for a row of elements may be enclosed in its own braces.  If an initialization list is present, we may omit the value of r.  If r \* c exceeds the number of values in the initialization list, the compiler initializes the uninitialized elements to 0.  If we omit the initialization list, we must specify r.  We must always specify c.

To pass a two-dimensional array to a function, we use the name of the array as the argument in the function call.  The corresponding function parameter receives the address of the array.  The parameter's declaration includes the array's column dimension.  For example,

|  |  |
| --- | --- |
| // Two-Dimensional Arrays  // pass2DArray.cpp  #include <iostream>  #define NCOLS 3  void foo(int a[][NCOLS], int r, int c);  int main () {  int a[2][NCOLS] = {{11, 12, 13}, {21, 22, 23}};  foo (a, 2, 3);  }  void foo(int a[][NCOLS], int r, int c) {  for (int i = 0; i < r; i++) {  for (int j = 0; j < c; j++)  std::cout << a[i][j] << ' ';  std::cout << std::endl;  }  } | 11 12 13  21 22 23 |

The compiler determines the start of each row within the array from the column dimension (NCOLS).  Since the compiler assumes row-major order, the function does not require the row dimension.

To work with a particular row from within a function, we pass the row as the argument in the function call.  We refer to a row of a two-dimensional array by the name of the array followed by the row number within brackets:

|  |
| --- |
| *identifier*[ *row* ] |

For example,

|  |  |
| --- | --- |
| // Two-Dimensional Arrays  // passRow.cpp  #include <iostream>  #define NCOLS 3  void foo(int a[], int c);  int main () {  int a[2][NCOLS] = {{11, 12, 13}, {21, 22, 23}};  foo (a[0], NCOLS); // pass first row  foo (a[1], NCOLS); // pass second row  }  void foo(int a[], int c) {  for (int i = 0; i < c; i++)  std::cout << a[i] << ' ';  std::cout << std::endl;  } | 11 12 13  21 22 23 |

a[0] points to the first row of a and holds the address of the first element of that row.  a[1] points to the second row of a and holds the address of the first element of that row.

Note the difference in type between a and &a[0][0].  Although both refer to the same address, a is of type int\*\* and &a[0][0] is of type int\*.

Run-Time Memory

Allocating a two-dimensional array at run-time involves five steps:

1. allocating stack memory for a pointer to a 1-D array of pointers to the rows of the 2-D array
2. allocating memory dynamically for a 1-D array of pointers to hold the addresses of the rows of the 2-D array
3. allocating memory dynamically for each row of the 2-D array
4. deallocating the dynamically allocated memory for each row of the 2-D array
5. deallocating the dynamically allocated memory for the array of pointers

For example,

|  |  |
| --- | --- |
| // Dynamically Allocated 2d Arrays  // dynamic2D.cpp  #include <iostream>  #include <iomanip>  int main () {  int c, r;  int \*\*a; // points to row addresses  std::cout << "Number of rows : ";  std::cin >> r;  std::cout << "Number of columns : ";  std::cin >> c;  a = new int\*[r]; // for row addresses  for (int i = 0; i < r; i++)  a[i] = new int[c]; // for elements  // of row i  for (int i = 0; i < r; i++)  for (int j = 0; j < c; j++)  a[i][j] = (i + 1) \* (j + 1);  std::cout << " ";  for (int j = 0; j < nc; j++)  std::cout << std::setw(4) << j + 1;  std::cout << std::endl;  for (int i = 0; i < r; i++) {  std::cout << std::setw(3) << i + 1;  for (int j = 0; j < c; j++)  std::cout << std::setw(4) << a[i][j];  std::cout << std::endl;  }  for (int i = 0; i < nr; i++)  delete [] a[i]; // deallocate row i  delete [] a; // deallocate pointers to rows  } | Number of rows : 4  Number of columns : 5  1 2 3 4 5  1 1 2 3 4 5  2 2 4 6 8 10  3 3 6 9 12 15  4 4 8 12 16 20 |

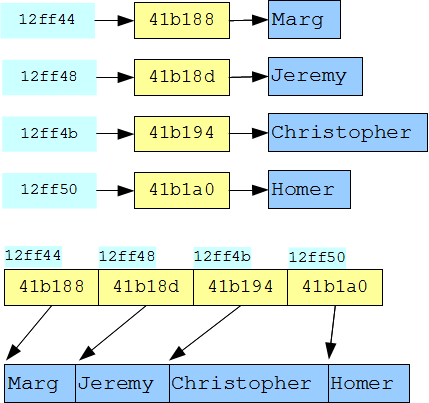
n-Dimensional Arrays

C++ does not restrict the number of dimensions in a multi-dimensional array.

The name of an n-dimensional array holds the address of the first (n-1)-dimensional (sub)array within the n-dimensional array.  For example, the name of a three-dimensional array holds the address of the first two-dimensional (sub)array within that three-dimensional array.  The name of a two-dimensional array holds the address of the first row of that array.  The name of a one-dimensional array holds the address of the first element of that array.

Ragged Arrays

Ragged arrays are commonly used to store sets of strings.  A *ragged* array is an array of addresses.  Each element of the array holds an address.  The data stored at each address may be of different size.  For example, C-style null terminated strings may differ in length, but the size of the pointer to each string is the same for every string.



The ragged array in the following program holds the list of names without wasting space

|  |  |
| --- | --- |
| // Ragged Arrays  // raggedArray.cpp  #include <iostream>  #include <iomanip>  int main () {  char \*name[] = {"Marg",  "Jeremy", "Christopher", "Homer"};  for (int i = 0; i < 4; i++)  std::cout << std::setw(7) << &name[i]  << std::setw(7) << std::hex <<  (int)&name[i][0] << std::dec  << ' ' << name[i] << std::endl;  std::cout << std::endl;  } | 12FF44 41b188 Marg  12FF48 41b18d Jeremy  12FF4C 41b194 Christopher  12FF50 41b1a0 Homer |

Each output line displays two addresses:

1. the address of the pointer to the string
2. the address of the string itself

Note that in this example each string is of a different length.

Pointer to a One-Dimensional Array

A pointer can hold the address of an array type.  The definition of a pointer to a one-dimensional array takes the form

|  |
| --- |
| *Type* (\**pointer*)[ *n* ]; // pointer to an array with n columns |

The parentheses in this definition distinguish the pointer to a 1-d array type from a definition of an array of pointers.  Parentheses are necessary since the bracket operator [] has higher precedence than the indirection operator \*.  Without the parentheses the compiler evaluates the brackets first and creates an array of pointers.

|  |
| --- |
| int (\*ptr)[ n ]; // pointer to a 1-d int array with n columns  int\* ptr [ n ]; // 1-d array of n pointers to int types |

The following program defines a pointer to a one-dimensional array of 3 ints, allocates memory for the array, initializes three elements in it, displays their values, and finally deallocates the memory:

|  |  |
| --- | --- |
| // Pointer to a 1D Array  // ptr\_1D\_Array.cpp  const int size = 3;  int main () {  int i;  int (\*ptr)[size] = (int(\*)[size]) new int[size];  (\*ptr)[0] = 10;  (\*ptr)[1] = 20;  (\*ptr)[2] = 30;  for (i = 0; i < size; i++)  std::cout << (\*ptr)[i] << ' ';  std::cout << std::endl;  delete [] ptr;  } | 10 20 30 |

Note the parentheses around the dereferenced pointer in the declaration, in the cast and in accessing the element values.

Pointer to a Two-Dimensional Array

A pointer can also hold the address of a two-dimensional array which has a fixed number of columns and dynamically allocated memory for a user-defined number of rows.

The following example defines a pointer to a two-dimensional array with 5 colums:

|  |  |
| --- | --- |
| // Pointer to a 2D Array  // ptr\_2D\_Array.cpp  #include <iostream>  #include <iomanip>  const int c = 5; // preset number of columns  int main () {  int r, i, j;  int (\*a)[c]; // pointer to a 2D array  std::cout << "Number of rows : ";  std::cin >> r;  a = new int[r][c]; // allocate array  for (i = 0; i < r; i++)  for (j = 0; j < c; j++)  a[i][j] = (i + 1) \* (j + 1);  std::cout << " ";  for (j = 0; j < c; j++)  std::cout << std::setw(4) << j + 1;  std::cout << std::endl;  for (i = 0; i < r; i++) {  std::cout << std::setw(3) << i + 1;  for (j = 0; j < c; j++)  std::cout << std::setw(4) << a[i][j];  std::cout << std::endl;  }  delete [] a; // deallocate array  } | Number of rows : 4  1 2 3 4 5  1 1 2 3 4 5  2 2 4 6 8 10  3 3 6 9 12 15  4 4 8 12 16 20 |

Note that we allocate memory for the two-dimensional array in a single statement and deallocate memory in a single statement as well.  A dynamically allocated array with a user-defined number of rows and a user-defined number of columns requires two steps for allocation and two for deallocation - one step in each involving an iteration.