Fundamental Types

Character Types

C++17 defines six character types:

* char
* signed char
* unsigned char
* wchar\_t
* char16\_t
* char32\_t

signed char, unsigned char, char

The first three char types occupy the same amount of memory - enough to hold the implementation's basic character set:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| char | | | | | | | |
| 1 Byte | | | | | | | |
|  |  |  |  |  |  |  |  |

The basic source-code character set consists of 96 characters:

* space
* control characters
  + horizontal tab
  + vertical tab
  + form feed
  + new-line
* a through z
* A through Z
* 0 1 2 3 4 5 6 7 8 9
* \_ { } [ ] # ( ) < > % : ; . ? \* + - / ^ & | ~ ! = , \ " '

The basic execution character set consists of these characters plus the following control characters

* null character - all bits set to 0
* alarm
* backspace
* carriage-return

You can find the escape sequences for the control characters listed in the [resources section](https://scs.senecac.on.ca/~chris.szalwinski/resources/escape.html).

The char type represents Unicode characters using the 8-bit Unicode Transmission Format (UTF-8).  UTF stands for Unicode Transformation Format.

Locales

C++17 uses locale objects for international portability.  A *locale* identifies the culture-specific set of features.

wchar\_t

The wchar\_t type is a wide character type that can store all the members of the largest character set amongst the supported locales.  The wchar\_t type has the same amount of memory and signedness as one of the other integral types.  The integral type associated with the wchar\_t type is called its *underlying type*.

We define a constant of wchar\_t type using the prefix L followed by the value in single quotes around character, octal, or hexadecimal notation:

|  |
| --- |
| wchar\_t k, m, n, p;  k = L'['; // character - note the leading L  m = L'\133'; // octal - note the leading L'\  n = L'\x5b'; // hexadecimal - note the leading L'\x  p = L'\X5B'; // hexadecimal - note the leading L'\X |

The single quotes identify a character literal.  The host platform converts this literal to its decimal value using the platform's encoding sequence.

char16\_t

The char16\_t type is a character type for representing characters using UTF-16.  UTF-16 is an encoding sequence that can hold 1,112,064 code points from 0 to 0x10FFFF.  This format is variable-length using one or two 16-bit code units; that is, it requires 16 or 32 bits to encode a character.  16 bits covers most of the Latin alphabets.  This format treats characters outside the Basic Multilingual Plane as a special case.  The Basic Multilingual Plane is the first group of continuous code points in Unicode, which contains the most commonly used characters.

The char16\_t type uses the same amount of memory and signedness as one of the other integral types.  The integral type associated with the char16\_t type is called its *underlying type*.

We define a constant of char16\_t type using the prefix u followed by single quotes around character, octal, or hexadecimal notation:

|  |
| --- |
| char16\_t k, m, n, p;  k = u'['; // character - note the leading u  m = u'\133'; // octal - note the leading u'\  n = u'\x5b'; // hexadecimal - note the leading u'\x  p = u'\X5B'; // hexadecimal - note the leading u'\X |

char32\_t

The char32\_t type is a character type for representing characters using UTF-32.  UTF-32 is a fixed-length character encoding that can hold code points from 0 to 0x7FFFFFFF.

This type has the same amount of memory and signedness as one of the other integral types.  The integral type associated with the char32\_t type is called its *underlying type*.

We define a constant of char32\_t type using the prefix U followed by single quotes around character, octal, or hexadecimal notation:

|  |
| --- |
| char32\_t k, m, n, p;  k = U'['; // character - note the leading U  m = U'\133'; // octal - note the leading U'\  n = U'\x5b'; // hexadecimal - note the leading U'\x  p = U'\X5B'; // hexadecimal - note the leading U'\X |

Void Type

The void type is an incomplete type.  A type is incomplete if it is missing some information.  C++ does not allow the creation of objects of type void.

We use a void type for the return type of functions that do not return a value and for generic pointers described in the next chapter.

Generic Pointer Type

The generic pointer type is distinct from any other pointer type defined in an application:  a pointer of generic type can hold the address of an object without holding the type information of that object itself.

A generic pointer type holds an address but is not associated with any object type.  The keyword void identifies a generic pointer type

|  |
| --- |
| void\* p; // generic pointer type |

Converting a pointer of any type into a generic type and vice versa does not lose its address:

|  |  |
| --- | --- |
| // Generic Pointer Type  // void.cpp  #include <iostream>  int main() {  int i;  void\* v = &i;  int\* j;  j = static\_cast<int\*>(v); // OK - j now holds the address of i  std::cout << &i << std::endl;  std::cout << j << std::endl;  } | 0024FA18  0024FA18 |

Converting from one pointer type to another requires an explicit cast:

|  |  |
| --- | --- |
| int\* i;  char\* c;  i = c; // ERROR - Incompatible  // Different Pointer Types | int\* i;  char\* c;  i = static\_cast<int\*>(  static\_cast<void\*>(c)); // OK |

Dereferencing a Generic Pointer

Since a generic pointer lacks the type information of the object pointed to, the compiler cannot dereference the pointer and identify the region of memory starting at the address pointed to.  To access the information stored at that address, the type of that information is required; we cast a generic pointer to the type that is associated with its address.

Hexadecimal Dump Example

The function hexDump() listed below displays the contents of a region of memory regardless of the type associated with that region.  The function receives the region's address and its size in number of bytes.  It casts the generic pointer a to a pointer to an unsigned char and displays the contents of c[i] in hexadecimal notation:

|  |  |
| --- | --- |
| // Hexadecimal Representation at an Address  // hexDump.cpp  #include <iostream>  #include <iomanip>  void hexDump(void\*, int);  int main() {  int i;  double x;  std::cout << "Integer value : ";  std::cin >> i;  std::cout << "is : ";  hexDump(&i, 4);  std::cout << std::endl;  std::cout << "Floating-point value : ";  std::cin >> x;  std::cout << "is : ";  hexDump(&x, 8);  std::cout << std::endl;  }  // Dump the first n bytes to the address a  //  void hexDump(void\* a, int n) {  unsigned char\* c =  static\_cast<unsigned char\*>(a);  std::cout.fill('0'); // zero fill  std::cout << std::hex; // hexadecimal output  for (int i = 0; i < n; i++)  std::cout << std::setw(2)  << (int)c[i] << ' ';  std::cout.fill(' '); // blank fill  std::cout << std::dec; // decimal output  } | Integer value : 2456  is : 98 09 00 00  Floating-point value : 4.56  is : 3d 0a d7 a3 70 3d 12 40 |

Note that in this example the platform is little-endian (little end first).

Array Types

An array type is a built-in type that consists of elements of identical type arranged contiguously in memory.  Each element is a subobject of the array type.  We declare an array type using the [] declarator operator.  An array can be constructed from any one of the

* fundamental types (except void)
* pointer types
* pointer to member types
* class types
* enumeration types (see the chapter entitled [Classes and Scoped Enumerations](../Week8/ClassEnumerations.docx))

Note that an array cannot be constructed directly from reference types.

One-Dimensional Array

The definition of a one-dimensional array takes one of the following forms

|  |
| --- |
| Type identifier[ c ]; // allocated on the stack  Type\* identifier = new Type[ n ]; // allocated on the heap |

Type is the type of each of the elements in the array.  c is the number of elements in the array and is an integer constant or constant integer expression.  n is the number of elements in the array and is an integer variable, integer constant or an integer constant expression.

Aggregate Initialization

We can initialize an array through *aggregate initilization*.  Aggregate initialization takes one of the following forms:

|  |
| --- |
| Type identifier[ c ] = { initializer-list };  Type identifier[ c ] = { };  Type identifier[ c ] { initializer-list };  Type identifier[ c ] { };  Type identifier[ ] = { initializer-list };  Type identifier[ ] { initializer-list };  Type\* identifier = new Type[ n ] { initializer-list };  Type\* identifier = new Type[ n ] { }; |

initializer-list is a comma-separated list of initial values.  If this 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.  If the braces are present but there is no initialization list, the compiler initializes all elements to 0.

For example, the program listed on the left produces the output listed on the right:

|  |  |
| --- | --- |
| // Aggregate Initialization  // initializers.cpp  #include <iostream>  int main() {  const int n = 6;  int a[] = { 1,2,3 };  int b[]{ 1,2,3 };  int c[5]{ 1,2,3 };  int d[5]{};  int\* f = new int[n]{ 1,2,3 };  int\* g = new int[n]{};  for (int e : a) // range-based for (see below)  std::cout << e;  std::cout << '|' << std::endl;  for (int e : b)  std::cout << e;  std::cout << '|' << std::endl;  for (int e : c)  std::cout << e;  std::cout << '|' << std::endl;  for (int e : d)  std::cout << e;  std::cout << '|' << std::endl;  for (int i = 0; i < n; ++i)  std::cout << f[i];  std::cout << '|' << std::endl;  for (int i = 0; i < n; ++i)  std::cout << g[i];  std::cout << '|' << std::endl;  delete[] f;  delete[] g;  } | 123|  123|  12300|  00000|  123000|  000000| |

Range-Based for

A range-based for is an iteration construct specifically designed for use with collections.  The collections can be of any form.  This construct steps sequentially through the elements of an array without requiring its size:

|  |  |
| --- | --- |
| // Range-Based for  // for\_each.cpp  #include <iostream>  int main () {  int a[]{1, 2, 3, 4, 5, 6};  for (int& e : a)  std::cout << e << ' ';  std::cout << std::endl;  } | 1 2 3 4 5 6 |

Range-Based for with Type Inference

A range-based for can infer the type of each element in the array from the array declaration itself:

|  |  |
| --- | --- |
| // Range-Based for  // for\_each\_auto.cpp  #include <iostream>  int main () {  int a[]{1, 2, 3, 4, 5, 6};  for (auto& e : a)  std::cout << e << ' ';  std::cout << std::endl;  } | 1 2 3 4 5 6 |