The C++ for OpenCL 1.0 and 2021 Programming Language Documentation

Khronos® OpenCL Working Group

Version DocRev2021.12, Mon, 20 Dec 2021 15:54:45 +0000: from git branch: commit: 3d36d9309228ef507d87b850a81e02d801fa7b05 tag: cxxforopencl-docrev2021.12

Table of Contents

Chapter 1. Introduction

This language is built on top of OpenCL C 3.0 unified and C++17 enabling most of regular C++ features in OpenCL kernel code. Most functionality from C++ and OpenCL C is inherited. Since both OpenCL C and C++ are derived from C and moreover C++ is almost fully backward compatible with C, the main design principle of C++ for OpenCL is to reapply existing OpenCL concepts to C++. Therefore, it is important to refer to The OpenCL Specification, Version 3.0 `section 3.2` and `section 3.3` detailing fundamental differences of OpenCL execution and memory models from the conventional C and C++ view.

This document describes the programming language in detail. It is not structured as a standalone document, but rather as an addition to OpenCL C 3.0 unified specification defined in The OpenCL C Specification, Version 3.0 and C++17 defined in The C++17 Specification. Where necessary this document refers to the specifications of those languages accordingly. A full understanding of C++ for OpenCL requires familiarity with the specifications or other documentation of both languages that C++ for OpenCL is built upon.

The description of C++ for OpenCL starts from highlighting *the differences to OpenCL C* and *the differences to C++*.

The majority of content covers the behavior that is not documented in the `OpenCL C 3.0 s6` and C++17 specifications. This is mainly related to interactions between OpenCL and C++ language features.

This document describes C++ for OpenCL language

- version 1.0 that is backward compatible with OpenCL 2.0; and
- version 2021 that is backward compatible with OpenCL 3.0.

C++ for OpenCL 2021 is described in this document as a provisional language version. While no large changes are envisioned in the future, some minor aspects might not remain identical in its final release.

Chapter 2. Version differences

The main difference between C++ for OpenCL version 1.0 and version 2021 comes from the difference between OpenCL 2.0 and OpenCL 3.0 with which they are respectively compatible. Support for some features of C++ for OpenCL 1.0 has become optional as described in OpenCL 3.0 s6.2.1. Predefined feature macros from OpenCL C 3.0 can be used to detect which optional features are present.

This impacts some C++ specific semantics mainly due to optionality of the generic address space (i.e. `__opencl_c_generic_address_space` feature) or program scope variables (i.e. `__opencl_c_program_scope_global_variables` feature).

Chapter 3. The C++ for OpenCL Programming Language

This programming language inherits features from `OpenCL C 3.0 s6` as well as C++17. Detailed aspects of OpenCL and C++ are not described in this document as they can be found in their official specifications.

This section documents various language features of C++ for OpenCL that are not covered in neither OpenCL nor C++ specifications, in particular:

- any behavior that deviates from C++17;
- any behavior that deviates from OpenCL C 2.0 or 3.0;
- any behavior that is not governed by OpenCL C and C++.

All language extensions to OpenCL C are applicable to C++ for OpenCL.

- Extensions to OpenCL C 2.0 or earlier versions apply to C++ for OpenCL version 1.0.
- Extensions to OpenCL C 3.0 or earlier versions except for OpenCL C 2.0, apply to C++ for OpenCL 2021.

3.1. Difference to C++

C++ for OpenCL supports the majority of standard C++17 features, however, there are some differences that are documented in this section.

3.1.1. Restrictions to C++ features

The following C++ language features are not supported:

- Virtual functions (C++17 `[class.virtual]`);
- References to functions (C++17 `[class.mfct]`);
- Pointers to class member functions (in addition to the regular non-member functions that are already restricted in OpenCL C);
- Exceptions (C++17 `[except]`);
- `dynamic_cast` operator (C++17 `[expr.dynamic.cast]`);
- Non-placement `new`/`delete` operators (C++17 `[expr.new]`/`[expr.delete]`);
- `thread_local` storage class specifier (C++17 `[basic.stc.thread]`);
- Standard C++ libraries (C++17 `[library]`).

Simultaneous initialization of static local objects performed by different work-items is not guaranteed to be free from race-conditions. Whether an implementation provides such a guarantee is indicated by the presence of the `__cpp_threadsafe_static_init` feature test macro[1].

The list above only contains extra restrictions that are not detailed in OpenCL C specification. As OpenCL restricts a number of C features, the same restrictions are inherited by C++ for OpenCL. The detailed list of C feature restrictions is provided in `OpenCL C 3.0 s6.11`.

3.2. Difference to OpenCL C

C++ for OpenCL provides backwards compatibility with OpenCL C for the majority of features. However, there are a number of exceptions that are described in this section. Some of them come from the nature of C++ but others are due to improvements in OpenCL features. Most of such improvements do not invalidate code written in OpenCL C, but simply provide extra functionality.

3.2.1. C++ related differences

C++ for OpenCL is a different language to OpenCL C and it is derived from C++ inheriting C++'s fundamental design principles. Hence C++ for OpenCL deviates from OpenCL C in the same areas where C++ deviates from C. This results in a more helpful language for developers and facilitates improvements in compilation tools without substantially increasing their complexity.

3.2.1.1. Implicit conversions

C++ is much stricter about conversions between types, especially those that are performed implicitly by the compiler. For example it is not possible to convert a `const` object to non-`const` implicitly. For details please refer to C++17 `[conv]`.

```
void foo(){
  const int *ptrconst;
  int *ptr = ptrconst; // invalid initialization discards const qualifier.
}
```

The same applies to narrowing conversions in initialization lists (C++17 `[dcl.init.list]`).

```
struct mytype {
 int i;
};
void foo(uint par){
  mytype var = {
    .i = par // narrowing from uint to int is disallowed.
  };
}
```

Some compilers allow silencing this error using a flag (e.g. in Clang `-Wno-error=c++11-narrowing` can be used).

Among other common conversions that will not be compiled in C++ mode there are pointer to integer or integer to pointer type conversions.

```
void foo(){
  int *ptr;
  int i = ptr; // incompatible pointer to integer conversion.
}
```

3.2.1.2. Null pointer constant

In C and OpenCL C the null pointer constant is defined using other language features as it is not represented explicitly i.e. commonly it is defined as

```
#define NULL ((void*)0)
```

In C++17 there is an explicit builtin pointer literal `nullptr` that should be used instead (C++17 `[lex.nullptr]`).

`NULL` macro definition in C++ for OpenCL follows C++17 `[support.types.nullptr]` where it is an implementation defined macro and it is not guaranteed to be the same as in OpenCL C. Reusing the definition of `NULL` from OpenCL C does not guarantee that any code with NULL is legal in C++ for OpenCL even if it is legal in OpenCL C.

```
#define NULL ((void*)0)
void foo(){
 int *ptr = NULL; // invalid initialization of int* with void*.
}
```

To improve code portability and compatibility, implementations are encouraged to define `NULL` as an alias to pointer literal `nullptr`.

3.2.1.3. Use of restrict

C++17 does not support `restrict` and therefore C++ for OpenCL can not support it either. Some compilers might provide extensions with some functionality of `restrict` in C++, e.g. `__restrict` in Clang.

This feature only affects optimizations and the source code can be modified by removing it. As a workaround to avoid manual modifications, macro substitutions can be used to either remove the keyword during the preprocessing by defining `restrict` as an empty macro or mapping it to another similar compiler features, e.g. `__restrict` in Clang. This can be done in headers or using `-D` compilation flag.

3.2.1.4. Limitations of goto statements

C++ is more restrictive with respect to entering the scope of variables than C. It is not possible to jump forward over a variable declaration statement apart from some exceptions detailed in C++17 `[stmt.dcl]`.

```
if (cond)
  goto label;
int n = foo();
label:  // invalid: jumping forward over declaration of n.
   // ...
```

3.2.1.5. Ternary selection operator

The ternary selection operator (`?:`) inherits its behaviour from both C++ and OpenCL C. It operates on three expressions `(exp1 ? exp2 : exp3)`. If all three expressions are scalar values, the C++17 rules for ternary operator are followed. If the result is a vector value, then this is equivalent to calling `select(exp3, exp2, exp1)` as described in `OpenCL C 3.0 s6.15.6`. The rules from OpenCL C impose limitation that `exp1` cannot be a vector of float values. However, `exp1` can be evaluated to a scalar float as it is contextually convertible to bool in C++.

3.2.2. OpenCL specific difference

This section describes where C++ for OpenCL differs from OpenCL C in OpenCL specific behavior.

3.2.2.1. Variadic macros

C++ for OpenCL eliminates the restriction on variadic macros from `OpenCL C 3.0 s6.11.f`. Variadic macros can be used normally as per C++17 `[cpp.replace]`.

3.2.2.2. Predefined macros

The macro `__OPENCL_C_VERSION__` described in `OpenCL C 3.0 s6.12`, is not defined.

The following new predefined macros are added in C++ for OpenCL:

- `__OPENCL_CPP_VERSION__` set to integer value reflecting the C++ for OpenCL version the translation unit is compiled for. The value `100` corresponds to the language version 1.0 and `202100` corresponds to the version 2021.
- `__CL_CPP_VERSION_1_0__` set to `100` and can be used for convenience instead of a literal.
- `__CL_CPP_VERSION_2021__` set to `202100` and can be used for convenience instead of a literal.

3.2.2.3. Atomic operations

C++ for OpenCL relaxes restriction from `OpenCL C 3.0 s6.15.12` to atomic types allowing them to be used by builtin operators, and not only by builtin functions. This relaxation does not apply to C++ for OpenCL version 2021 if the sequential consistency memory model (i.e. `__opencl_c_atomic_order_seq_cst` feature) is not supported.

Operators on atomic types behave as described in C++17 sections `[atomics.types.int]` `[atomics.types.pointer]` `[atomics.types.float]`.

```
// Assumes support of sequential consistency memory model.
atomic_int acnt;
acnt++; // equivalent to atomic_fetch_add(&acnt, 1);
```

3.2.2.4. Use of Blocks

Blocks that are defined in `OpenCL C 3.0 s6.14` are not supported and their use might be replaced by lambdas (C++17 `[expr.prim.lambda]`) in future versions.

The above implies that builtin functions using blocks, such as `enqueue_kernel`, are not supported in C++ for OpenCL.

3.3. Address spaces

C++ for OpenCL inherits address space behavior from OpenCL C 3.0 s6.7.

This section only documents behavior related to C++ features. For example, conversion rules are extended from the qualification conversion in C++17 [conv.qual] but the compatibility is determined using notation of sets and overlapping of address spaces from section 5.1.3 of The Embedded C Specification. For OpenCL kernel languages there are two main semantics depending on whether generic address space (OpenCL C 3.0 s6.7.5) is supported on not. The generic address space is always supported for

- C++ for OpenCL 1.0;
- C++ for OpenCL 2021 with the presence of __opencl_c_generic_address_space feature as explained in OpenCL C 3.0 s6.2.1.

If generic address space is not supported, qualification conversions with pointer type where address spaces differ are not allowed. If generic address space is supported, implicit conversions are allowed from a named address space (except for __constant) to generic address space. The reverse conversion is only allowed explicitly. The __constant address space does not overlap with any other, therefore, no valid conversion between __constant and any other address space exists. This is aligned with rules from OpenCL C 3.0 s6.7.9 and this logic regulates semantics described in this section.

3.3.1. Casts

C-style casts follow rules of OpenCL C 3.0 s6.7.9. Conversions of references and pointers to the generic address space can be done by any C++ cast operator (as an implicit conversion); converting from generic to named address space can only be done using the dedicated addrspace_cast operator. The addrspace_cast operator can only convert between address spaces for pointers and references and no other conversions are allowed to occur. Note that conversions between __constant and any other other address space are disallowed.

```
// Example assumes generic address space support.
int * genptr; // points to generic address space.

// generic -> named address space conversions.
__private float * ptrfloat = reinterpret_cast<__private float*>(genptr); // illegal.
__private float * ptrfloat = addrspace_cast<__private float*>(genptr); // illegal.
__private int * ptr = addrspace_cast<__private int*>(genptr); // legal.

// named -> generic address space conversion.
float * genptrfloat = reinterpret_cast<float*>(ptr); // legal.

// disjoint address space conversion.
__constant int * constptr = addrspace_cast<__constant int*>(genptr); // illegal.
```

If generic address space is not supported, any conversion of references/pointers pointing to different address spaces is illegal.

```
// Example without generic address space support.
int * privptr; // points to private address space.

// The same address space conversions.
__private float * ptrfloat = reinterpret_cast<__private float*>(privptr); // legal.
__private float * ptrfloat = addrspace_cast<__private float*>(privptr); // illegal.
__private int * ptr = addrspace_cast<__private int*>(privptr); // legal, no op.
float * privptrfloat = reinterpret_cast<float*>(ptr); // legal.

// disjoint address space conversion.
__constant int * constptr = addrspace_cast<__constant int*>(privptr); // illegal.
```

3.3.2. References

Reference types can be qualified with an address space.

```
__private int & ref = ...; // references int in __private address space.
```

By default references refer to generic address space objects if generic address space is supported or private address space otherwise, except for dependent types that are not template specializations (see *Address space inference*).

```
int & ref = ...; // references int in generic address space if it is
                 // supported otherwise in __private address space.
```

Address space compatibility checks are performed when references are bound to values. The logic follows the rules from address space pointer conversion (OpenCL C 3.0 s6.7.9).

```
void f(float &ref, __global float &globref) {
  const int& tmp = ref; // legal - reference to generic/__private address space object
                        // can bind to a temporary object created in __private
                        // address space.

  __global const int& globtmp = globref; // error: reference to global address space
                                         // object cannot bind to a temporary object
                                         // created in __private  address space.
}
```

3.3.3. Address space inference

This section details what happens if address spaces for types are not provided in the source code explicitly. Most of the logic for address space inference (i.e. default address space) follows rules from OpenCL C 3.0 s6.7.8.

References inherit rules from pointers and therefore refer to generic address space objects by

default (see *References*) if generic address space is supported, otherwise they refer to private address space objects.

Class static data members are deduced to `__global` address space. Note, that if the C++ for OpenCL 2021 implementation does not support program scope variables (i.e. `__opencl_c_program_scope_global_variables` feature from `OpenCL C 3.0 s6.1` is unsupported) the address space qualifier must be specified explicitly and it must be the `__constant` address space.

All non-static member functions take an implicit object parameter `this` that is a pointer type. By default the `this` pointer parameter is in the generic address space if it is supported and in the private address space otherwise. All concrete objects passed as an argument to the implicit `this` parameter will be converted to this default (generic or private) address space first if such conversion is valid. Therefore, when member functions are called with objects created in disjoint address spaces from the default one, the compilation must fail. To prevent the failure the address space on implicit object parameter `this` must be specified using address space qualifiers on member functions (see *Member function qualifier*). For example, use of member functions with objects in `__constant` address space will always require a `__constant` member function qualifier as `__constant` address space is disjoint with any other.

Member function qualifiers can also be used in case address space conversions are undesirable for example for performance reasons. For example, a method can be implemented to exploit memory access coalescing for segments with memory bank.

Address spaces are not deduced for:

- non-pointer/non-reference template parameters except for template specializations or non-type based template parameters.
- non-pointer/non-reference class members except for static data members that are deduced to the `__global` address space for C++ for OpenCL 1.0 or C++ for OpenCL 2021 with the `__opencl_c_program_scope_global_variables` feature.
- non-pointer/non-reference type alias declarations.
- decltype expressions.

```
template <typename T>
void foo() {
  T m; // address space of 'm' will be known at template instantiation time.
  T * ptr; // 'ptr' points to generic address space object when it is
           // supported otherwise to __private address space.
  T & ref = ...; // 'ref' references an object in generic address space when
                 // it is supported otherwise in __private address space.
};

template <int N>
struct S {
  int i; // 'i' has no address space.
  static int ii; // 'ii' is in global address space if program scope variables
                 // are supported; otherwise this statement is not legal.
  int * ptr; // 'ptr' points to int in generic address space if it is supported;
             // otherwise to __private address space.
  int & ref = ...; // 'ref' references int in generic address space if it is
                   // supported; otherwise in __private address space.
};

template <int N>
void bar()
{
  S<N> s; // 's' is in __private address space.
}
```

```
struct c1 {};
using alias_c1 = c1; // 'alias_c1' is 'c1'.
using alias_c1_ptr = c1 *; // 'alias_c1_ptr' is a generic address space pointer to
                           // 'c1' when generic address space is supported; otherwise
                           // it points to 'c1' located in __private address space.
```

```
__kernel void foo()
{
  __local int i;
  decltype(i)* ii; // type of 'ii' is '__local int *__private'.
}
```

For the placeholder type specifier `auto` an address space of the outer type is deduced as if it would be any other regular type. However if `auto` is used in a reference or pointer type, the address space of a pointee is taken from the type of the initialization expression. The logic follows rules for `const` and `volatile` qualifiers.

```
// This example assumes that generic address space is supported.
__kernel void foo()
{
  __local int i;
  constexpr int c = 1;

  __constant auto cai = c; // type of 'cai' is '__constant int' (no deduction).

  auto aii = cai; // type of 'aii' is '__private int' (regular deduction).

  auto *ptr = &i; // type of 'ptr' is '__local int * __private'
                  // (addr space of a pointer is deduced regularly,
                  // addr space of its pointee is taken from 'i').

  auto *&refptr = ptr; // type of 'refptr' is '__local int * generic & __private'
                       // (addr space of a reference and type of referencing object
                       // is deduced regularly,
                       // addr space of a pointee is taken from the pointee of 'ptr').
}
```

3.3.4. Member function qualifier

C++ for OpenCL allows specifying an address space qualifier on member functions to signal that they are to be used with objects constructed in a specific address space. This works just the same as qualifying member functions with `const` or any other qualifiers. The overloading resolution will select the candidate with the most specific address space if multiple candidates are provided. If there is no conversion to an address space among candidates, compilation will fail with a diagnostic.

```
struct C {
  C() __local {};
  C() __private {};
  constexpr C() __constant {};

  void foo() __local;
  void foo(); // This is implicitly qualified by generic address space
              // if it is supported otherwise by '__private'.
};

__kernel void bar() {
  __local C c1;      // will resolve to the first constructor overload.
  __private C c2;    // will resolve to the second constructor overload.
  __constant C c3{}; // will resolve to the third constructor overload.
  c1.foo(); // will resolve to the first 'foo'.
  c2.foo(); // will resolve to the second 'foo'.
  c3.foo(); // error due to mismatching address spaces - can't convert to
            // '__local' or generic/'__private' address spaces.
}
```

All member functions can be qualified by an address space qualifier including constructors and destructors.

3.3.5. Lambda function

The address space qualifier can be optionally added for lambda expressions after the attributes. Similar to method qualifiers, they will alter the default address space of lambda call operator that has generic address space by default if it is supported otherwise private address space.

```
__kernel void foo() {
  auto priv1 = []() __private {};
  priv1();
  auto priv2 = []() __global {};
  priv2(); // error: lambda object and its expression have mismatching address space.
  __constant auto const3 = []() __constant{};
  const3();

  [&] () __global {} (); // error: lambda temporary is in __private address space.

  [&] () mutable __private {} ();
  [&] () __private mutable {} (); // error: mutable specifier should precede address
                                  // space.
}
```

3.3.6. Implicit special members

The prototype for implicit special members (default, copy or move constructor, copy or move assignment, destructor) has the default address space for an implicit object pointer and reference parameters (see also *Member function qualifier*). This default address space is generic if it is supported or private otherwise.

```
class C {
  // Has the following implicitly defined member functions.

  // C(); /* implicit 'this' parameter is a pointer to */
          /* object in generic address space if supported, or */
          /* in private address space otherwise. */

  // C(const C & par); /* 'this'/'par' is a pointer/reference to */
                       /*  object in generic address space */
                       /*  if supported, or */
                       /* in private address space otherwise. */

  // C(C && par); /* 'this'/'par' is a pointer/r-val reference to */
                 /* object in generic address space if supported, or */
                 /* in private address space otherwise. */

  // C & operator=(const C & par); /* 'this'/'par'/return value is */
                                   /* a pointer/reference/reference to */
                                   /* object in generic address space */
                                   /* if supported, or */
                                   /* in private address space otherwise. */

  // C & operator=(C && par)'; /* 'this'/'par'/return value is */
                               /* a pointer/r-val reference/reference to */
                               /* object in generic address space, */
                               /* if supported, or */
                               /* in private address space otherwise. */
};
```

3.3.7. Builtin operators

All builtin operators are available with the specific address spaces, thus no address space conversions (i.e. to generic address space) are performed.

3.3.8. Templates

There is no deduction of address spaces in non-pointer/non-reference template parameters and dependent types (see *Address space inference*). The address space of a template parameter is deduced during type deduction if it is not explicitly provided in the instantiation.

```
1  template<typename T>
2  void foo(T* i){
3    T var;
4  }
5
6  __global int g;
7  void bar(){
8    foo(&g); // error: template instantiation failed as function scope variable 'var'
9             // appears to be declared in __global address space (see line 3).
10 }
```

It is not legal to specify multiple different address spaces between template definition and instantiation. If multiple different address spaces are specified in a template definition and instantiation, compilation of such a program will fail with a diagnostic. This restriction immediately follows from OpenCL C 3.0 s6.7 that disallows multiple address space qualifiers on a type.

```
template <typename T>
void foo() {
  __private T var;
}

void bar() {
  foo<__global int>(); // error: conflicting address space qualifiers are provided
                       // for 'var', '__global' and '__private'.
}
```

Once a template has been instantiated, regular restrictions for address spaces will apply as described in OpenCL C 3.0 s6.7.

```
template<typename T>
void foo(){
  T var;
}

void bar(){
  foo<__global int>(); // error: function scope variable 'var' cannot be declared
                       // in '__global' address space.
}
```

3.3.9. Temporary materialization

All temporaries are materialized in __private address space. If a reference with another address space is bound to them, a conversion will be generated in case it is valid, otherwise compilation will fail with a diagnostic.

```
int bar(const unsigned int &i); // references generic address space object
                                // if generic address space is supported
                                // otherwise private address space object.

void foo() {
  bar(1); // temporary is created in __private address space but (if generic
          // address space is supported) converted to generic address space
          // of parameter reference.
}

void f(__global float &ref) {
  __global const int& newref = ref; // error: address space mismatch between
                                    // temporary object created to hold value
                                    // converted float->int and local variable
                                    // (can't convert from __private to __global).
}
```

3.3.10. Construction, initialization and destruction

Construction, initialization and destruction of objects in `__private` and `__global` address space follow the general principles of C++. For program scope objects or static objects in the function scope with non-trivial constructors and destructors, the implementation defines an ABI format for runtime initialization and destruction of global objects before/after all kernels are enqueued.

Objects in `__local` address space can not have initializers in declarations and therefore a constructor can not be called. All objects created in the local address space have undefined state at the point of their declaration. Developers are free to define a special member function that can initialize local address space objects after their declaration. Any default values provided for the initialization of members in a class declaration are ignored when creating the local address space objects. Since the initialization is performed after the variable declaration, special handling is required for classes with data members that are references because their values can not be overwritten trivially. Destructors of local address space objects are not invoked automatically. They can be called manually.

```
class C {
  int m;
// If generic address space is not supported or for performance optimization
// purposes the following members might be required.
public:
  __local C & operator=(const C & par) __local;
  ~C() __local;
};
__kernel void foo() {
  __local C locobj{}; // error: local address space objects can't be initialized
  __local C locobj; // uninitialised object.
  locobj = {}; // calling copy assignment operator is allowed.
  locobj.~C(); // local address space object destructors are not invoked
               // automatically.
}
```

User defined constructors in `__constant` address space must be `constexpr`. Such objects can be initialized using literals and initialization lists if they do not require any user defined conversions.

Objects in `__constant` address space can be initialized using:

- Literal expressions;
- Uniform initialization syntax `{}`;
- Using implicit constructors.
- Using `constexpr` constructors.

```
struct C1 {
  int m;
};

struct C2 {
  int m;
  constexpr C2(int init) __constant : m(init) {};
};

__constant C1 c1obj1 = {1};
__constant C1 c1obj2 = C1();
__constant C2 c2obj1(1);
```

Non-trivial destructors for objects in non-default address spaces (i.e. all other than generic address space when it is supported or `__private` otherwise) are not required to be supported by implementations. The macro `__opencl_cpp_destructor_with_address_spaces`, which is defined if and only if such destructors are supported by an implementation, can be used to check whether this functionality can be used in kernel sources. Additionally destructors with global objects might not be supported even if address spaces are supported with destructors in general. Such functionality is indicated by the presence of the `__opencl_cpp_global_destructor` macro. If the macro `__opencl_cpp_global_destructor` is defined then `__opencl_cpp_destructor_with_address_spaces` must

also be defined.

Note that the destruction of objects in named address spaces `__global`, `__local`, or `__private` can be performed using destructors with default address space (i.e. generic) by utilising address space conversions.

```
1  // Example assumes generic address space support.
2  class C {
3  public:
4  #ifdef __opencl_cpp_destructor_with_address_spaces
5    ~C() __local;
6  #else
7    ~C();
8  #endif
9  };
10
11 kernel void foo() {
12   __local C locobj;
13   locobj.~C();    // uses destructor in local address space (on line 5)
14                   // if such destructors are supported,
15                   // otherwise uses generic address space destructor (on line 7)
16                   // converting to generic address prior to call into destructor.
17 }
```

However, when generic address space feature is unsupported, absence of destructor support with address spaces results in a compilation failure when such destructors overloaded with non-default address spaces are encountered in the kernel code.

```
// Example assumes generic address space is not supported.
class C {
public:
  ~C();
};

kernel void foo() {
  __local C locobj;
  locobj.~C();    // error due to illegal conversion of 'this' from __local
                  // to __private address space pointer.
}
```

3.3.11. Nested pointers

C++ for OpenCL does not allow implicit address space conversions in nested pointers even with compatible address spaces. The following rules apply when converting between address spaces in nested pointers:

- Implicit conversions of address spaces in nested pointers are disallowed.
- Any address space conversion in nested pointers with safe casts (e.g. `const_cast`, `static_cast`,

`addrspace_cast`) is disallowed.

- Any address space conversion in nested pointers can be done using low level C-style or `reinterpret_cast`. No compatibility check is performed for address spaces in nested pointers.

```
local int * * locdefptr;
constant int * * cnstdefptr;
int * * defdefptr;
defdefptr = const_cast<int * *>(locdefptr); // illegal.
defdefptr = static_cast<int * *>(cnstdefptr); // illegal.
defdefptr = addrspace_cast<int * *>(cnstdefptr); // illegal.
defdefptr = reinterpret_cast<int * *>(locdefptr); // legal.
defdefptr = reinterpret_cast<int * *>(cnstdefptr); // legal.
```

3.3.12. Address space removal type trait

`template<class T> struct __remove_address_space;`

C++ for OpenCL 2021 supports the type trait `__remove_address_space` that provides the member `typedef type` which is the same as `T`, except that its topmost address space qualifier is removed. Its effect is analogous to `remove_const` and other similar type traits in C++17 `[meta.trans]`. The trait only removes an address space qualifier from a given type, therefore, all other type qualifiers such as `const` or `volatile` remain unchanged.

```
template<typename T>
void foo(T *par) {
  T var1; // error: function scope variable cannot be declared in global
          // address space.
  __private T var2; // error: conflicting address space qualifiers are provided
                    // between types '__private T' and '__global int'.
  __private __remove_address_space<T>::type var3; // type of var3 is __private int.
}

void bar() {
  __global int* ptr;
  foo(ptr);
}
```

3.4. C++ casts

C++ has three cast operators in addition to C-style casts. Additional logic specific to address spaces are applied to all casts as detailed in *conversions with address spaces*. `reinterpret_cast` has some additional functionality:

- Conversion between vectors and scalars are allowed.
- Conversion between OpenCL types are disallowed.

3.4.1. Vectors and scalars

`reinterpret_cast` reinterprets between integral types like integers and pointers. In C++ for openCL this also includes vector types, and so using `reinterpret_cast` between vectors and scalars is also possible, as long as the size of the vectors are the same.

```
int i;
short2 s2 = reinterpret_cast<short2>(i); // legal.
int2 i2 = reinterpret_cast<int2>(i); // illegal.

short8 s8;
int4 i4 = reinterpret_cast<int4>(s8); // legal.
long l4 = reinterpret_cast<long>(s8); // illegal.
```

3.4.2. OpenCL types

Some of the OpenCL-specific types, defined as "Other Built-in Data Types" in `OpenCL C 3.0 s6.3.3`, are convertible to integer literals, but since they are not conceptually integral, they can not be used with `reinterpret_cast`. Therefore conversions of an OpenCL-specific type to any distinct type are illegal.

```
queue_t q;
reserve_id_t id = reinterpret_cast<reserve_id_t>(q); // illegal.
int i = reinterpret_cast<int>(id); // illegal.
```

3.5. Kernel functions

Kernel functions have implicit C linkage (C++17 [dcl.link]) which means that C++ specific features are not supported. Therefore, the kernel functions:

- Can not be class members (C++17 [class.mfct]);
- Can not be overloaded (C++17 [over]);
- Can not be function templates (C++17 [temp.fct]).

Moreover the types used in parameters of the kernel functions must be:

- Trivial and standard-layout types C++17 [basic.types] (plain old data types) for parameters passed by value;
- Standard-layout types for pointer parameters. The same applies to references[2] if an implementation supports them in kernel parameters.

These are additional restrictions to the list detailed in OpenCL C 3.0 s6.11.

[1] The macro belongs to the list of C++20's feature test macros.

[2] Whether C++ features (e.g references) can be used in functions with C linkage is implementation-defined (C++17 [dcl.link]).

Chapter 4. Normative References

1. “The OpenCL Specification, Version 3.0”, https://www.khronos.org/registry/OpenCL/.
2. “The OpenCL C Specification, Version 3.0”, https://www.khronos.org/registry/OpenCL/.
3. “ISO/IEC 14882:2017 - Programming languages — C++”, https://www.iso.org/standard/68564.html. References are to sections of this specific version, referred to as the “The C++17 Specification”, although other versions exist.
4. “ISO/IEC TR 18037:2008 Programming languages - C - Extensions to support embedded processors”, https://www.iso.org/standard/51126.html. References are to sections of this specific version, referred to as the “The Embedded C Specification”, although other versions exist.

Acknowledgements

The C++ for OpenCL documentation is the result of the contributions of many people. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

- Anastasia Stulova, Arm
- Neil Hickey, Arm
- Sven van Haastregt, Arm
- Marco Antognini, Arm
- Kevin Petit, Arm
- Stuart Brady, Arm
- Ole Strøhm, Arm
- Justas Janickas, Arm