



The C++ for OpenCL Programming Language Documentation (Draft)

Khronos® OpenCL Working Group

Version v3.0.1-Provisional-2-g297653b, Mon, 27 Apr 2020 16:47:09 +0000: from git branch: commit:
297653bd8fdda852925d796567e71fadbae4b712

Table of Contents

1. Introduction	2
2. The C++ for OpenCL Programming Language	3
2.1. Difference to C++	4
2.1.1. Restrictions to C++ features	4
2.2. Difference to OpenCL C	5
2.2.1. C++ related differences	5
2.2.2. OpenCL specific difference	6
2.3. Address spaces	8
2.3.1. Casts	8
2.3.2. References	8
2.3.3. Deduction & Default address space	8
2.3.4. Member function qualifier	10
2.3.5. Lambda function	11
2.3.6. Implicit special members	11
2.3.7. Builtin operators	12
2.3.8. Templates	12
2.3.9. Temporary materialization	13
2.3.10. Construction, initialization and destruction	13

Copyright 2019-2020 The Khronos Group.

Khronos licenses this file to you under the Creative Commons Attribution 4.0 International (CC BY 4.0) License (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at <https://creativecommons.org/licenses/by/4.0/>

Unless required by applicable law or agreed to in writing, material distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. If all or a portion of this material is re-used, notice substantially similar to the following must be included:

This documentation includes material developed at The Khronos Group (<http://www.khronos.org/>). Khronos supplied such material on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, under the terms of the Creative Commons Attribution 4.0 International (CC BY 4.0) License (the "License"), available at <https://creativecommons.org/licenses/by/4.0/>. All use of such material is governed by the term of the License. Khronos bears no responsibility whatsoever for additions or modifications to its material.

Khronos is a registered trademark, and OpenCL is a trademark of Apple Inc. and used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

Chapter 1. Introduction

This language is built on top of OpenCL C v2.0 and C++17 enabling most of regular C++ features in OpenCL kernel code. Most functionality from OpenCL C and C++ is inherited.

This document describes the programming language in details. It is not structured as a standalone document, but rather an addition to [OpenCL C v2.0 s6](#) and C++17 ([ISO/IEC 14882:2017](#)). 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 v2.0 section 6 and C++17 specifications. This is mainly related to interactions between OpenCL and C++ language features.

Chapter 2. The C++ for OpenCL Programming Language

This programming language inherits features from [OpenCL C v2.0 s6](#) as well as C++17 ([ISO/IEC 14882:2017](#)). 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 either OpenCL or C++ specifications, in particular:

- any behavior that deviates from C++17;
- any behavior that deviates from OpenCL C v2.0;
- any behavior that is not governed by OpenCL C and C++.

2.1. 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.

2.1.1. Restrictions to C++ features

The following C++ language features are not supported:

- Virtual functions
- Exceptions
- `dynamic_cast` operator
- Non-placement `new/delete` operators
- Standard C++ libraries

2.2. 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 old code, but simply provide extra functionality.

2.2.1. 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.

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
}
```

Null literal

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

C++ for OpenCL does not define `NULL` and therefore any source code using it must be modified to use `nullptr` instead. However as a workaround to avoid large modifications `NULL` can also be defined/aliased to `nullptr` in custom headers or using command line flag `-D`. It is not recommended to reuse the C definition of `NULL` in C++ for OpenCL as it may cause compilation failures in cases that work for C.

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

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.

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  
    // ...
```

2.2.2. OpenCL specific difference

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

Variadic macros

C++ for OpenCL eliminates the restriction on variadic macros from [OpenCL C v2.0 s6.9.e](#). Variadic macros can be used normally as per C++17 [\[cpp.replace\]](#).

Predefined macros

The predefined macros `OPENCL_C_VERSION` and `NULL` (see also [Null literal](#)), described in [OpenCL C v2.0 s6.10](#), are not supported.

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

- `OPENCL_CPP_VERSION` set to value `100`.
- `CL_CPP_VERSION_1_0` also set to `100` and can be used for convenience instead of a literal.

Atomic operations

C++ for OpenCL relaxes restriction from [OpenCL C v2.0 s6.13.11](#) to atomic types allowing them to be used by builtin operators, and not only by builtin functions.

Operators on atomic types behave as described in C++17 sections [\[atomics.types.int\]](#) [\[atomics.types.pointer\]](#) [\[atomics.types.float\]](#).

```
atomic_int acnt;  
acnt++; // equivalent to atomic_fetch_add(&acnt, 1);
```

Use of Clang Blocks

Clang Blocks that are defined by the Objective-C language are not supported and their use can be replaced by lambdas (C++17 [\[expr.prim.lambda\]](#)).

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

2.3. Address spaces

C++ for OpenCL inherits address space behavior from [OpenCL C v2.0 s6.5](#).

This section only documents behavior related to C++ features. For example conversion rules are extended from the qualification conversion but the compatibility is determined using notation of sets and overlapping of address spaces from Embedded C ([ISO/IEC JTC1 SC22 WG14 N1021 s3.1.3](#)). For OpenCL it means that implicit conversions are allowed from a named address space (except for `constant`) to generic ([OpenCL C v2.0 6.5.5](#)). The reverse conversion is only allowed explicitly. The `constant` address space does not overlap with any other and therefore no valid conversion between `__constant` and any other address space exists. Most of the rules follow this logic.

2.3.1. Casts

C-style casts follow rules of [OpenCL C v2.0 s6.5.5](#). Conversions to generic address space can be made 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. Operator `addrspace_cast` can only convert between address spaces and no other conversions are allowed to occur. Note that conversions between `__constant` and any other other address space are disallowed.

```
int * genptr;
__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.
float * genptrfloat = reinterpret_cast<float*>(ptr); // legal.
__constant int * constptr = addrspace_cast<__constant int*>(genptr); // illegal.
```

2.3.2. 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, except for dependent types that are not template specializations (see [Deduction](#)). Address space compatibility checks are performed when references are bound to values. The logic follows the rules from address space pointer conversion ([OpenCL v2.0 s6.5.5](#)).

2.3.3. Deduction & Default address space

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 deduction (i.e. default address space) follows rules from [OpenCL 2.0 s6.5](#).

References inherit rules from pointers and therefore refer to generic address space objects by default (see [References](#)).

Class static data members are deduced to `__global` 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. All concrete objects passed as an argument to the implicit `this` parameter will be converted to the generic address space first if such conversion is valid. Therefore programs using objects in the `constant address space` will not be compiled unless the address space is explicitly specified using address space qualifiers on member functions (see [Member function qualifier](#)) as the conversion between `constant` and `generic` is disallowed. Member function qualifiers can also be used in case conversion to the generic address space is undesirable (even if it is legal). For example, a method can be implemented to exploit memory access coalescing for segments with memory bank. This not only applies to regular member functions but to constructors and destructors too.

Address spaces are not deduced for:

- non-pointer/non-reference template parameters or any dependent types except for template specializations.
- non-pointer/non-reference class members except for static data members that are deduced to `__global` address space.
- non-pointer/non-reference 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.
    T & ref = ...; // 'ref' references an object in generic address space.
};

template <int N>
struct S {
    int i; // 'i' has no address space.
    static int ii; // 'ii' is in global address space.
    int * ptr; // 'ptr' points to int in generic address space.
    int & ref = ...; // 'ref' references int in generic 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'.
```

```
__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.

```
__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 * & __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').
}
```

2.3.4. 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 {
    void foo() __local;
    void foo();
};

__kernel void bar() {
    __local C c1;
    C c2;
    __constant C c3;
    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.
}

```

2.3.5. 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.

```

__kernel void test_qual() {
    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.
}

```

2.3.6. Implicit special members

All implicit special members (default, copy or move constructor, copy or move assignment, destructor) will be generated with the generic address space.

```
class C {
    // Has the following implicitly defined member functions
    // void C() /*__generic*/;
    // void C(const /*__generic*/ C &) /*__generic*/;
    // void C(/*__generic*/ C &&) /*__generic*/;
    // operator= '/*__generic*/ C &(/*__generic*/ C &&)' ;
    // operator= '/*__generic*/ C &(const /*__generic*/ C &) /*__generic*/;
}
```

2.3.7. Builtin operators

All builtin operators are available in the specific address spaces, thus no conversion to generic address space is performed.

2.3.8. Templates

There is no deduction of address spaces in non-pointer/non-reference template parameters and dependent types (see [Deduction](#)). 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
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.

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

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

Once a template has been instantiated, regular restrictions for address spaces will apply.

```

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

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

```

2.3.9. 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);

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

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

```

2.3.10. 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, the implementation (i.e. compiler) defines an ABI format for 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. Classes that are instantiated with local address space objects can not have references as their members. The same restriction applies recursively to the members and base classes. Destructors of local address space objects are not invoked automatically. They can be called manually if required.

User defined constructors are not allowed to construct objects in `__constant` address space. 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.

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