Type punning in modern C++

version 1.1

Timur Doumler

@timur_audio

C++ Russia
30 October 2019

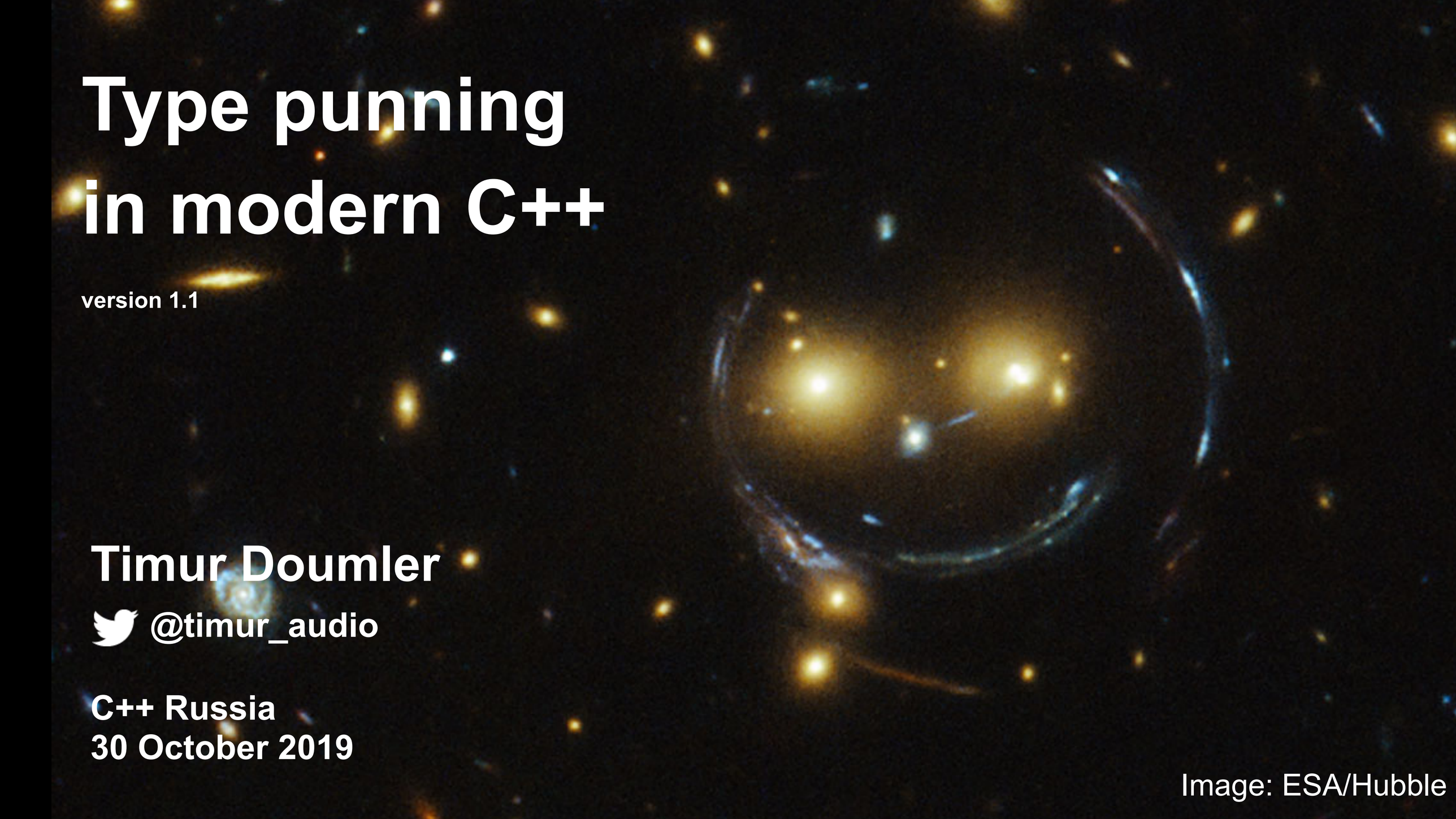


Image: ESA/Hubble

Type punning in modern C++

How not to do type punning in modern C++

How to write portable code that achieves the same effect as type punning in modern C++

How to write portable code that achieves the same effect as type punning in modern C++ without causing undefined behaviour

pun[1]

/pʌn/

noun

a joke exploiting the different possible meanings of a word or the fact that there are words which sound alike but have different meanings.
"the Railway Society reception was an informal party of people of all stations (excuse the pun) in life"
synonyms: play on words, wordplay, double entendre, double meaning, innuendo, witticism, quip;
More

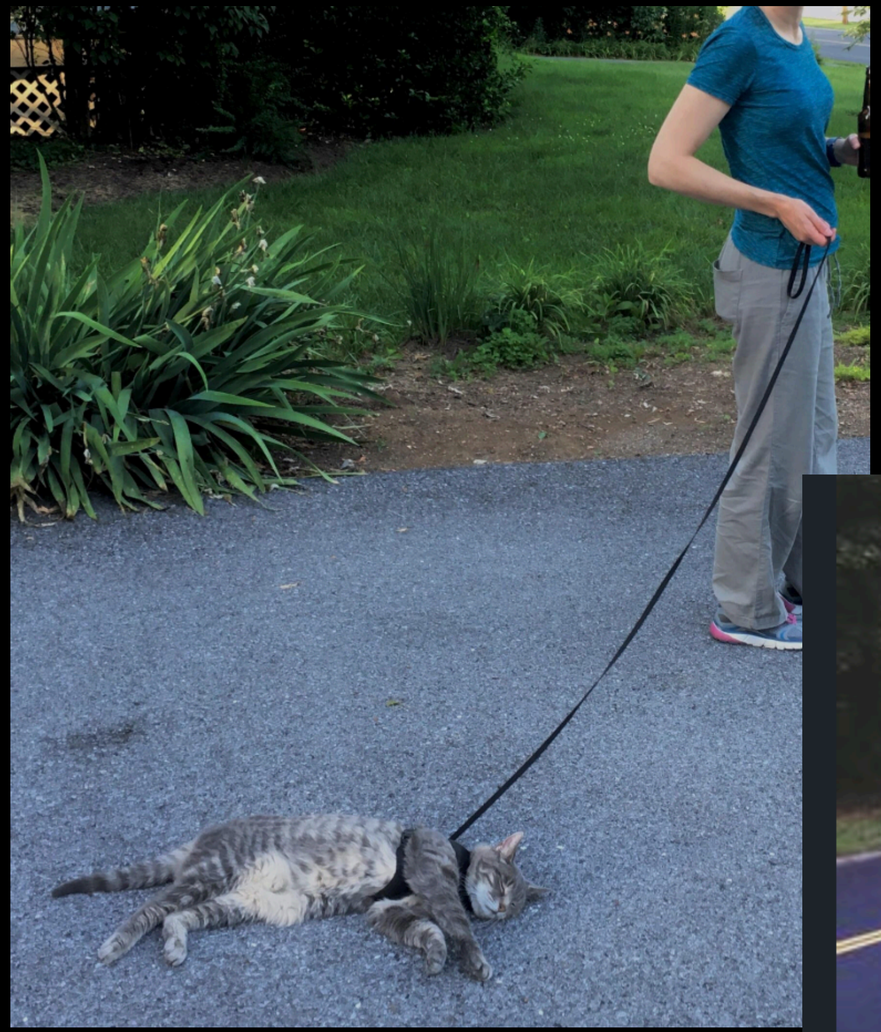
verb

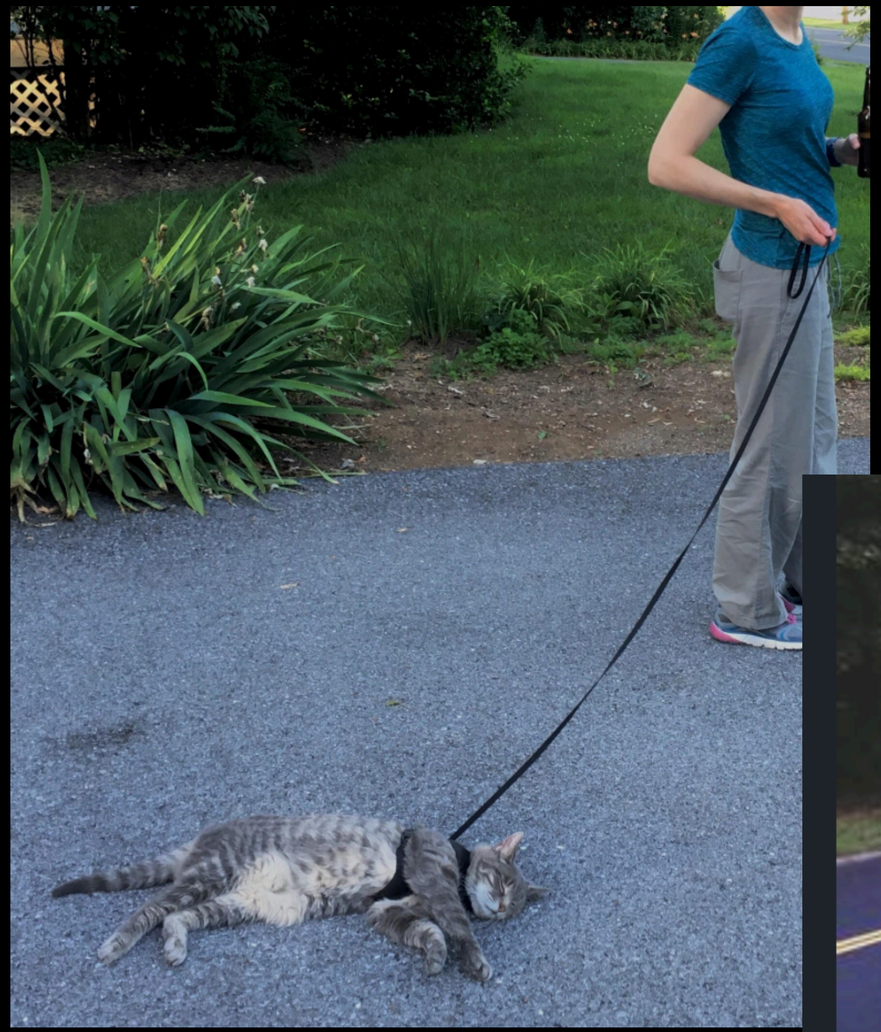
make a pun.
"Freeth adopted the nickname Free in punning allusion to his beliefs"

Type punning

In computer science, type punning is a common term for any programming technique that subverts or circumvents the type system of a programming language in order to achieve an effect that would be difficult or impossible to achieve within the bounds of the formal language. Wikipedia

()





```
struct Widget {
    virtual void doSomething() {
        printf("Widget");
    }
};

struct Gizmo {
    virtual void doSomethingCompletelyDifferent() {
        printf("Gizmo");
    }
};

int main() {
    Gizmo g;
    Widget* w = (Widget*)&g;
    w->doSomething();
}
```

Example from **Luna**

@lunasorcery

```
struct Widget {
    virtual void doSomething() {
        printf("Widget");
    }
};

struct Gizmo {
    virtual void doSomethingCompletelyDifferent() {
        printf("Gizmo");
    }
};

int main() {
    Gizmo g;
    Widget* w = (Widget*)&g;
    w->doSomething();
}
```

Example from **Luna**

@lunasorcery

4 The conversions performed by

(4.1) — a `const_cast`,

(4.2) — a `static_cast`,

(4.3) — a `static_cast` followed by a `const_cast`,

(4.4) — a `reinterpret_cast`, or

(4.5) — a `reinterpret_cast` followed by a `const_cast`,

can be performed using the cast notation of explicit type conversion. The same semantic restrictions and behaviors apply, with the exception that in performing a `static_cast` in the following situations the conversion is valid even if the base class is inaccessible:

(4.6) — a pointer to an object of derived class type or an lvalue or rvalue of derived class type may be explicitly converted to a pointer or reference to an unambiguous base class type, respectively;

(4.7) — a pointer to member of derived class type may be explicitly converted to a pointer to member of an unambiguous non-virtual base class type;

(4.8) — a pointer to an object of an unambiguous non-virtual base class type, a glvalue of an unambiguous non-virtual base class type, or a pointer to member of an unambiguous non-virtual base class type may be explicitly converted to a pointer, a reference, or a pointer to member of a derived class type, respectively.

If a conversion can be interpreted in more than one of the ways listed above, the interpretation that appears first in the list is used, even if a cast resulting from that interpretation is ill-formed. If a conversion can be interpreted in more than one way as a `static_cast` followed by a `const_cast`, the conversion is ill-formed. [*Example:*

Don’t use C-style casts.

```
struct Widget {
    virtual void doSomething() {
        printf("Widget");
    }
};

struct Gizmo {
    virtual void doSomethingCompletelyDifferent() {
        printf("Gizmo");
    }
};

int main() {
    Gizmo g;
    Widget* w = (Widget*)&g;
    w->doSomething();
}
```

Example from **Luna**

@lunasorcery

C++

```
1  #include <cstdio>
2
3  struct Widget {
4      virtual void doSomething() {
5          printf("Widget");
6      }
7  };
8
9  struct Gizmo {
10     virtual void doSomethingDifferent() {
11         printf("Gizmo");
12     }
13 };
14
15 int main() {
16     Gizmo g;
17     Widget* w = (Widget*)&g;
18     w->doSomething();
19 }
20
```

x86-64 clang (trunk)

-O3

A | 11010 | ./a.out | .LX0: | lib.f: | .text

```
1  main:
2          push    rax
3          mov     edi, offset .L.str
4          xor     eax, eax
5          call    printf
6          xor     eax, eax
7          pop     rcx
8          ret
9  .L.str:
10         .asciz  "Gizmo"
```

C++

```
1   #include <cstdio>
2
3   struct Widget {
4       virtual void doSomething() {
5           printf("Widget");
6       }
7   };
8
9   struct Gizmo {
10      virtual void doSomethingDifferent() {
11          printf("Gizmo");
12      }
13  };
14
15  int main() {
16      Gizmo g;
17      Widget* w = (Widget*)&g;
18      w->doSomething();
19  }
20
```

x86-64 gcc (trunk)

-O3

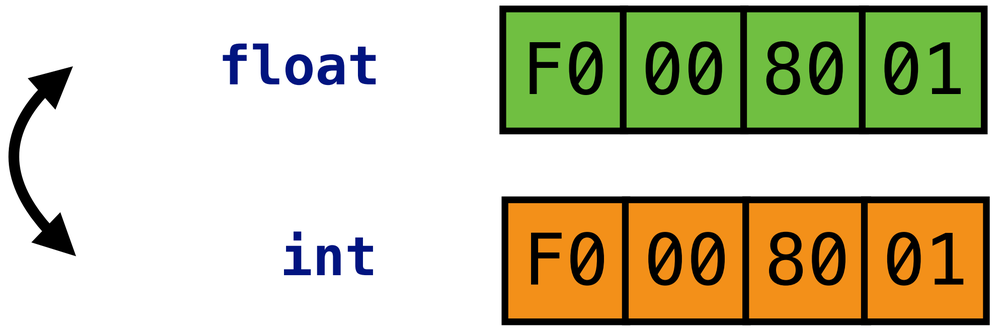
A 11010 ./a.out .LX0: lib.f:

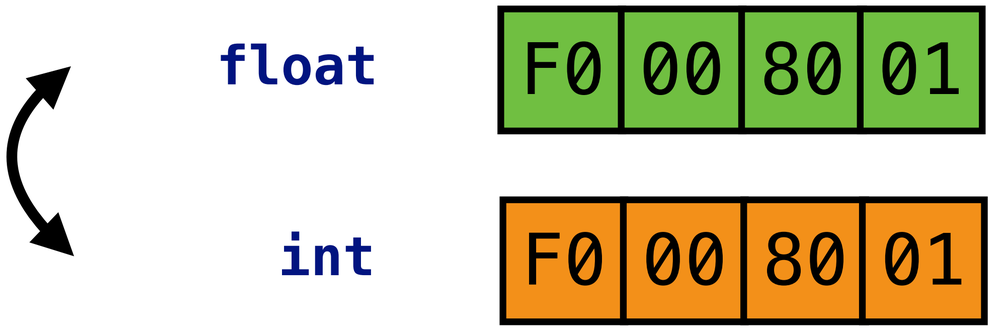
```
1   main:
```

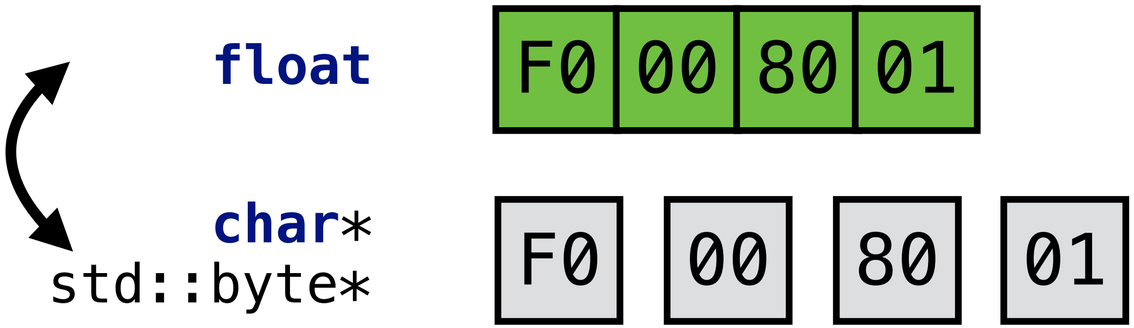
CppCon 2018: Barbara Geller & Ansel Sermersheim “Undefined Behavior is Not an Error”

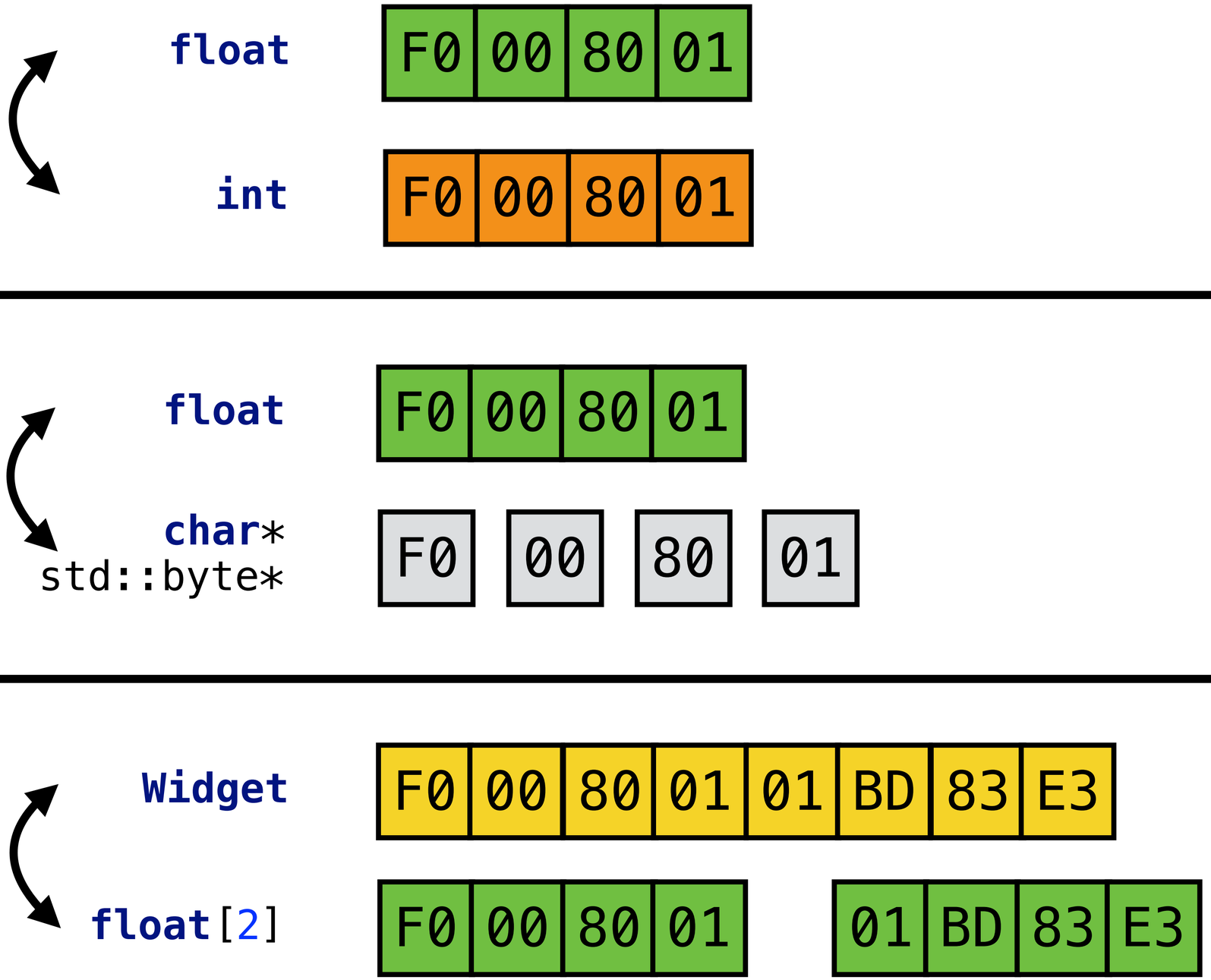
2,889 views

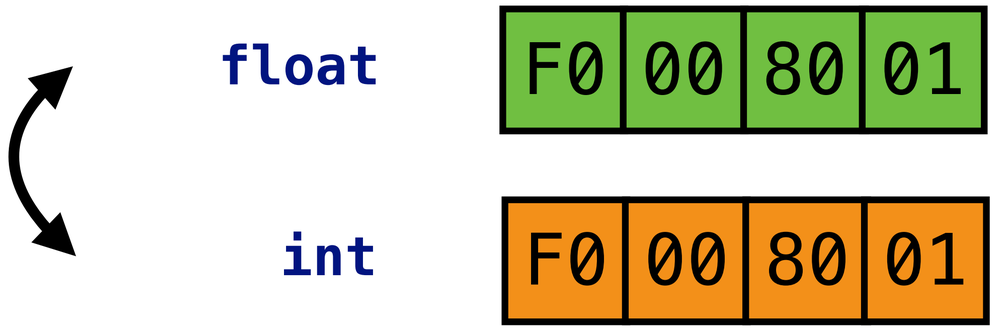
49 10 SHARE SAVE


float
F0 00 80 01
int
F0 00 80 01


float
F0 00 80 01
int
F0 00 80 01


float
F0 00 80 01
char*
std::byte*
F0
00
80
01


float
F0 00 80 01
int
F0 00 80 01
float
F0 00 80 01
char*
std::byte*
F0 00 80 01
Widget
F0 00 80 01 01 BD 83 E3
float[2]
F0 00 80 01
01 BD 83 E3


float
F0 00 80 01
int
F0 00 80 01

float
F0 00 80 01
char*
std::byte*
F0 00 80 01

Widget
F0 00 80 01 01 BD 83 E3
float[2]
F0 00 80 01
01 BD 83 E3

```
float Q_rsqrt( float number )
{
    int i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( int * ) &y;                        // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the ****?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
```

Fast inverse square root from Quake III Arena (1999, original code w/ comments)

```
float Q_rsqrt( float number )
{
    int i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( int * ) &y;                        // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the ****?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
```

Fast inverse square root from Quake III Arena (1999, original code w/ comments)

11 If a program attempts to access the stored value of an object through a glvalue whose type is not similar ([conv.qual]) to one of the following types the behavior is undefined:[52]

(11.1) — the dynamic type of the object,

(11.2) — a type that is the signed or unsigned type corresponding to the dynamic type of the object, or

(11.3) — a `char`, `unsigned char`, or `std::byte` type.

If a program invokes a defaulted copy/move constructor or copy/move assignment operator for a union of type `U` with a glvalue argument that does not denote an object of type *cv* `U` within its lifetime, the behavior is undefined. [*Note:* Unlike in C, C++ has no accesses of class type. — *end note*]

52) The intent of this list is to specify those circumstances in which an object may or may not be aliased. □

```
float Q_rsqrt( float number )
{
    const float x2 = number * 0.5F;
    const float threehalfs = 1.5F;

    union {
        float f;
        int i;
    } conv = {number}; // member 'f' set to value of 'number'.

    conv.i  = 0x5f3759df - ( conv.i >> 1 );
    conv.f  *= ( threehalfs - ( x2 * conv.f * conv.f ) );

    return conv.f;
}
```

```
float Q_rsqrt( float number )
{
    const float x2 = number * 0.5F;
    const float threehalfs = 1.5F;

    union {
        float f;
        int i;
    } conv = {number}; // member 'f' set to value of 'number'.

    conv.i  = 0x5f3759df - ( conv.i >> 1 );
    conv.f  *= ( threehalfs - ( x2 * conv.f * conv.f ) );

    return conv.f;
}
```

1 In a union, a non-static data member is *active* if its name refers to an object whose lifetime has begun and has not ended ([basic.life]). At most one of the non-static data members of an object of union type can be active at any time, that is, the value of at most one of the non-static data members can be stored in a union at any time. [*Note:* One special guarantee is made in order to simplify the use of unions: If a standard-layout union contains several standard-layout structs that share a common initial sequence ([class.mem]), and if a non-static data member of an object of this standard-layout union type is active and is one of the standard-layout structs, it is permitted to inspect the common initial sequence of any of the standard-layout struct members; see [class.mem]. — *end note*]

2 The size of a union is sufficient to contain the largest of its non-static data members. Each non-static data member is allocated as if it were the sole member of a non-union class. [*Note:* A union object and its non-static data members are pointer-interconvertible ([basic.compound], [expr.static.cast]). As a consequence, all non-static data members of a union object have the same address. — *end note*]

11 If a program attempts to access the stored value of an object through a glvalue whose type is not similar ([conv.qual]) to one of the following types the behavior is undefined:[52]

(11.1) — the dynamic type of the object,

(11.2) — a type that is the signed or unsigned type corresponding to the dynamic type of the object, or

(11.3) — a `char`, `unsigned char`, or `std::byte` type.

If a program invokes a defaulted copy/move constructor or copy/move assignment operator for a union of type U with a glvalue argument that does not denote an object of type *cv* U within its lifetime, the behavior is undefined. [*Note:* Unlike in C, C++ has no accesses of class type. — *end note*]

52) The intent of this list is to specify those circumstances in which an object may or may not be aliased. □

Don’t use unions.

Use std::variant.

You can’t do type punning in C++.


Drive on left
Tenez la gauche
Links fahren

…but why?

…but why?

- aliasing rules
- object lifetime rules
- alignment rules
- rules for valid value representations

…but why?

- aliasing rules
- object lifetime rules
- alignment rules
- rules for valid value representations

11 If a program attempts to access the stored value of an object through a glvalue whose type is not similar ([conv.qual]) to one of the following types the behavior is undefined:[52]

(11.1) — the dynamic type of the object,

(11.2) — a type that is the signed or unsigned type corresponding to the dynamic type of the object, or

(11.3) — a `char`, `unsigned char`, or `std::byte` type.

If a program invokes a defaulted copy/move constructor or copy/move assignment operator for a union of type `U` with a glvalue argument that does not denote an object of type *cv* `U` within its lifetime, the behavior is undefined. [*Note:* Unlike in C, C++ has no accesses of class type. — *end note*]

52) The intent of this list is to specify those circumstances in which an object may or may not be aliased. □

11 If a program attempts to access the stored value of an object through a glvalue whose type is not similar ([conv.qual]) to one of the following types the behavior is undefined:[52]

(11.1) — the dynamic type of the object,

(11.2) — a type that is the signed or unsigned type corresponding to the dynamic type of the object, or

(11.3) — a `char`, `unsigned char`, or `std::byte` type.

If a program invokes a defaulted copy/move constructor or copy/move assignment operator for a union of type `U` with a glvalue argument that does not denote an object of type *cv* `U` within its lifetime, the behavior is undefined. [*Note:* Unlike in C, C++ has no accesses of class type. — *end note*]

52) The intent of this list is to specify those circumstances in which an object may or may not be aliased. □

```
void test(int* a, int* b)
{
    *a = 1;
    *b = 2;
    *a += *b;
}
```

```
void test(int* a, int* b)
{
    *a = 1;
    *b = 2;
    *a += *b;
}

int main()
{
    int x = 0;
    int y = 0;

    test(&x, &y);
    return x;
}
```

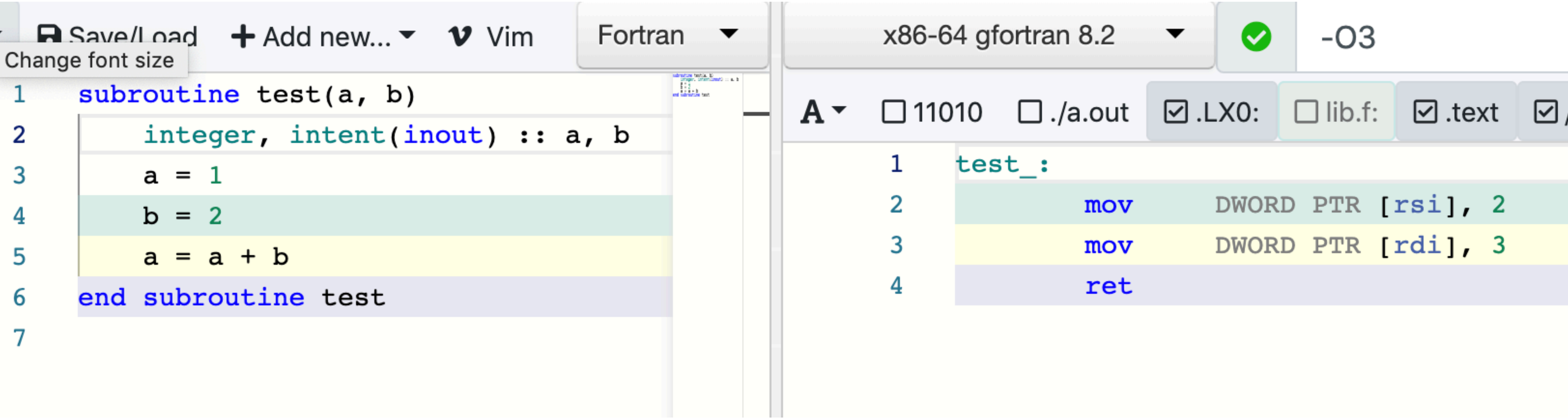
```
void test(int* a, int* b)
{
    *a = 1;
    *b = 2;
    *a += *b;
}

int main()
{
    int x = 0;
    int y = 0;

    test(&x, &y);
    return x;    // a == 3
}
```

```
void test(int* a, int* b)
{
    *a = 1;
    *b = 2;
    *a += *b;
}

int main()
{
    int x = 0;


    test(&x, &x);
    return x;
}
```

```
void test(int* a, int* b)
{
    *a = 1;
    *b = 2;
    *a += *b;
}

int main()
{
    int x = 0;


    test(&x, &x);
    return x;       // a == 4
}
```

```
void test(int* a, int* b)    // b is aliasing a. OK because same type
{
    *a = 1;
    *b = 2;
    *a += *b;
}

int main()
{
    int x = 0;


    test(&x, &x);  // OK
    return x;      // a == 4
}
```


C++
x86-64 gcc 9.2
-O3
11010
./a.out
.LX0:
lib.f:
.text
1 void test(int* a, int* b)
2 {
3 *a = 1;
4 *b = 2;
5 *a += *b;
6 }
1 test(int*, int*):
2 mov DWORD PTR [rdi], 1
3 mov DWORD PTR [rsi], 2
4 add DWORD PTR [rdi], 2
5 ret


Save/Load
Add new...
Vim
Fortran
Change font size
subroutine test(a, b)
integer, intent(inout) :: a, b
a = 1
b = 2
a = a + b
end subroutine test
x86-64 gfortran 8.2
-O3
11010
./a.out
.LX0:
lib.f:
.text
test_:
mov DWORD PTR [rsi], 2
mov DWORD PTR [rdi], 3
ret

```
void test(int* a, float* b)     // b CANNOT be aliasing a.
{
    *a = 1;
    *b = 2;
    *a += static_cast<int> (*b);
}
```

```
void test(int* a, float* b)    // b CANNOT be aliasing a.
{
    *a = 1;
    *b = 2;
    *a += static_cast<int> (*b);
}

int main()
{
    int x = 0;

    test(&x, reinterpret_cast<float*>(&x));  // Undefined behaviour!
    return x;       // x == 💩
}
```

shafik / WhatIsStrictAliasingAndWhyDoWeCare.md

Subscribe | Star 133 | Fork 10

Last active 10 days ago • Report abuse

Code | Revisions 44 | Stars 133 | Forks 10

Embed <script src="https://gi | Download ZIP

What is Strict Aliasing and Why do we Care?

WhatIsStrictAliasingAndWhyDoWeCare.md | Raw

What is the Strict Aliasing Rule and Why do we care?

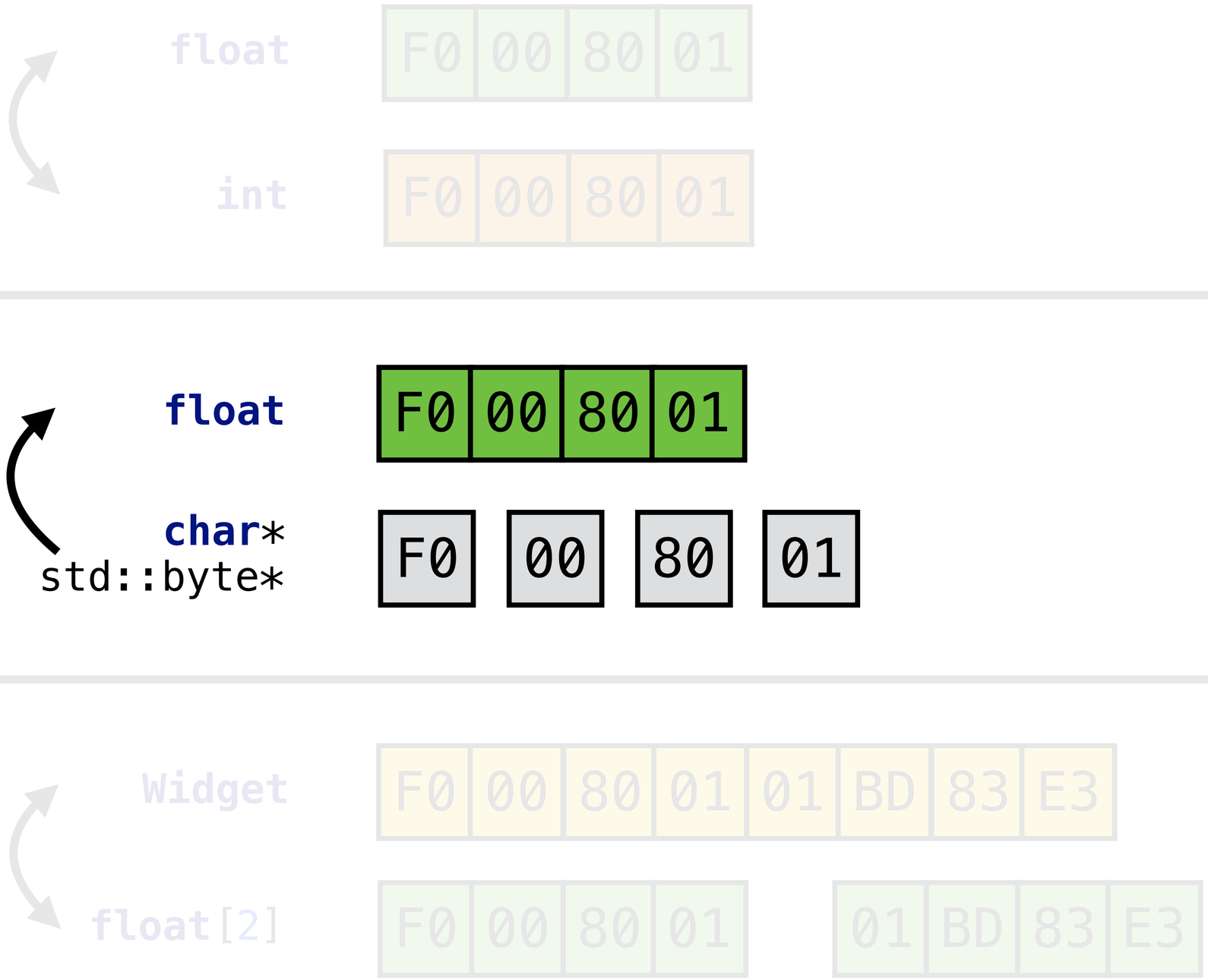
(OR Type Punning, Undefined Behavior and Alignment, Oh My!)

What is strict aliasing? First we will describe what is aliasing and then we can learn what being strict about it means.

In C and C++ aliasing has to do with what expression types we are allowed to access stored values through. In both C and C++ the standard specifies which expression types are allowed to alias which types. The compiler and optimizer are allowed to assume we follow the aliasing rules strictly, hence the term *strict aliasing rule*. If we attempt to access a value using a type not allowed it is classified as undefined behavior(**UB**). Once we have undefined behavior all bets are off, the results of our program are no longer reliable.

Unfortunately with strict aliasing violations, we will often obtain the results we expect, leaving the possibility the a future version of a compiler with a new optimization will break code we thought was valid. This is undesirable and it is a worthwhile goal to understand the strict aliasing rules and how to avoid violating them.

- aliasing rules
- object lifetime rules
- alignment rules
- rules for valid value representations


float
F0 00 80 01
int
F0 00 80 01
float
F0 00 80 01
char*
std::byte*
F0 00 80 01
Widget
F0 00 80 01 01 BD 83 E3
float[2]
F0 00 80 01
01 BD 83 E3

```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    X* p = (X*)malloc(sizeof(struct X));
    p->a = 1;
    p->b = 2;
    return p;
}
```

```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    X* p = (X*)malloc(sizeof(struct X));
    p->a = 1;
    p->b = 2;
    return p;
}
```

1 The *lifetime* of an object or reference is a runtime property of the object or reference. A variable is said to have *vacuous initialization* if it is default-initialized and, if it is of class type or a (possibly multi-dimensional) array thereof, that class type has a trivial default constructor. The lifetime of an object of type `T` begins when:

(1.1) — storage with the proper alignment and size for type `T` is obtained, and

(1.2) — its initialization (if any) is complete (including vacuous initialization) ([dcl.init]),

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union ([dcl.init.aggr], [class.base.init]), or as described in [class.union]. The lifetime of an object *o* of type `T` ends when:

(1.3) — if `T` is a non-class type, the object is destroyed, or

(1.4) — if `T` is a class type, the destructor call starts, or

(1.5) — the storage which the object occupies is released, or is reused by an object that is not nested within *o* ([intro.object]).

6 Before the lifetime of an object has started but after the storage which the object will occupy has been allocated[30] or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that represents the address of the storage location where the object will be or was located may be used but only in limited ways. For an object under construction or destruction, see [class.cdtor]. Otherwise, such a pointer refers to allocated storage ([basic.stc.dynamic.allocation]), and using the pointer as if the pointer were of type `void*` is well-defined. Indirection through such a pointer is permitted but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:

(6.1) — the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a *delete-expression*,

(6.2) — the pointer is used to access a non-static data member or call a non-static member function of the object, or

(6.3) — the pointer is implicitly converted ([conv.ptr]) to a pointer to a virtual base class, or

(6.4) — the pointer is used as the operand of a `static_cast` ([expr.static.cast]), except when the conversion is to pointer to *cv* `void`, or to pointer to *cv* `void` and subsequently to pointer to *cv* `char`, *cv* `unsigned char`, or *cv* `std::byte` ([cstddef.syn]), or

(6.5) — the pointer is used as the operand of a `dynamic_cast` ([expr.dynamic.cast]).

7 Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. For an object under construction or destruction, see [class.cdtor]. Otherwise, such a glvalue refers to allocated storage ([basic.stc.dynamic.allocation]), and using the properties of the glvalue that do not depend on its value is well-defined. The program has undefined behavior if:

(7.1) — the glvalue is used to access the object, or

(7.2) — the glvalue is used to call a non-static member function of the object, or

(7.3) — the glvalue is bound to a reference to a virtual base class ([dcl.init.ref]), or

(7.4) — the glvalue is used as the operand of a `dynamic_cast` ([expr.dynamic.cast]) or as the operand of `typeid`.

8 If, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object, if:

7 Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. For an object under construction or destruction, see [class.cdtor]. Otherwise, such a glvalue refers to allocated storage ([basic.stc.dynamic.allocation]), and using the properties of the glvalue that do not depend on its value is well-defined. The program has undefined behavior if:

(7.1) — the glvalue is used to access the object, or

(7.2) — the glvalue is used to call a non-static member function of the object, or

(7.3) — the glvalue is bound to a reference to a virtual base class ([dcl.init.ref]), or

(7.4) — the glvalue is used as the operand of a `dynamic_cast` ([expr.dynamic.cast]) or as the operand of `typeid`.

8 If, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object, if:

```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    X* p = (X*)malloc(sizeof(struct X));   // UB: there is no object of type X here!
    p->a = 1;
    p->b = 2;
    return p;
}
```

std::launder is unrelated to this.
It does not help here.

std::launder is for pointers
to actual existing, alive objects.

6.6.2 Object model [intro.object]

1 The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is created by a definition, by a *new-expression*, when implicitly changing the active member of a union, or when a temporary object is created ([conv.rval], [class.temporary]). An object occupies a region of storage in its period of construction ([class.cdtor]), throughout its lifetime, and in its period of destruction ([class.cdtor]). [*Note:* A function is not an object, regardless of whether or not it occupies storage in the way that objects do. — *end note*] The properties of an object are determined when the object is created. An object can have a name. An object has a storage duration which influences its lifetime. An object has a type. Some objects are polymorphic ([class.virtual]); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the *expressions* ([expr.compound]) used to access them.

```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    X* p = (X*)malloc(sizeof(struct X));   // UB: there is no object of type X here!
    p->a = 1;
    p->b = 2;
    return p;
}
```

```
void process(Stream* stream)
{
    std::unique_ptr<char[]> buffer = stream->read();

    if (buffer[0] == WIDGET)
        processWidget(reinterpret_cast<Widget*>(buffer.get()));
    else
        // ...
}
```

```
void process(Stream* stream)
{
    std::unique_ptr<char[]> buffer = stream->read();

    if (buffer[0] == WIDGET)
        processWidget(reinterpret_cast<Widget*>(buffer.get()));    // UB :(
    else
        // ...
}
```

…so how do we fix this?

```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    X* p = (X*)malloc(sizeof(struct X));   // UB: there is no object of type X here!
    p->a = 1;
    p->b = 2;
    return p;
}
```

```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    char storage[sizeof(X)];
    X* p = new(storage) X;
    p->a = 1;
    p->b = 2;
    return p;
}
```

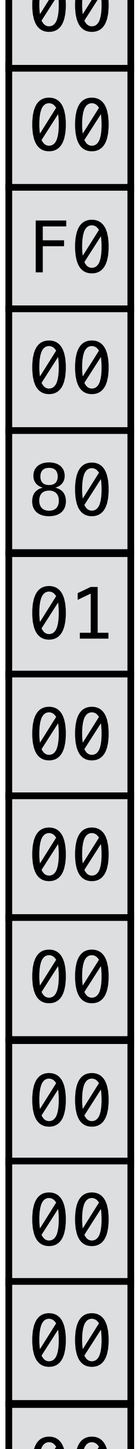
```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    char storage[sizeof(X)];
    X* p = new(storage) X;  // char array “provides storage” for object p
    p->a = 1;
    p->b = 2;
    return p;
}
```

```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    char storage[sizeof(X)];
    X* p = new(storage) X;
    p->a = 1;
    p->b = 2;
    return p;
}
```

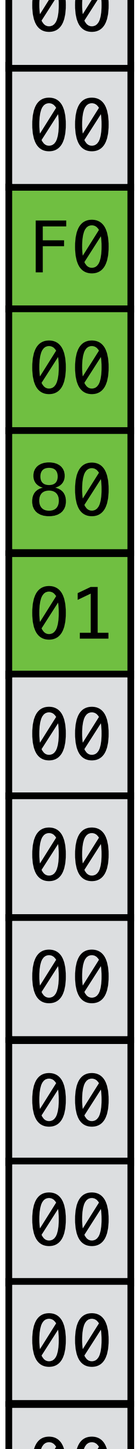
```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    char storage[sizeof(X)];
    X* p = new(storage) X;  // Undefined behaviour :(
    p->a = 1;
    p->b = 2;
    return p;
}
```

…but why?

- aliasing rules
- object lifetime rules
- alignment rules
- rules for valid value representations


00
00
F0
00
80
01
00
00
00
00
00
00
00
00

float x



6.6.6 Alignment [basic.align]

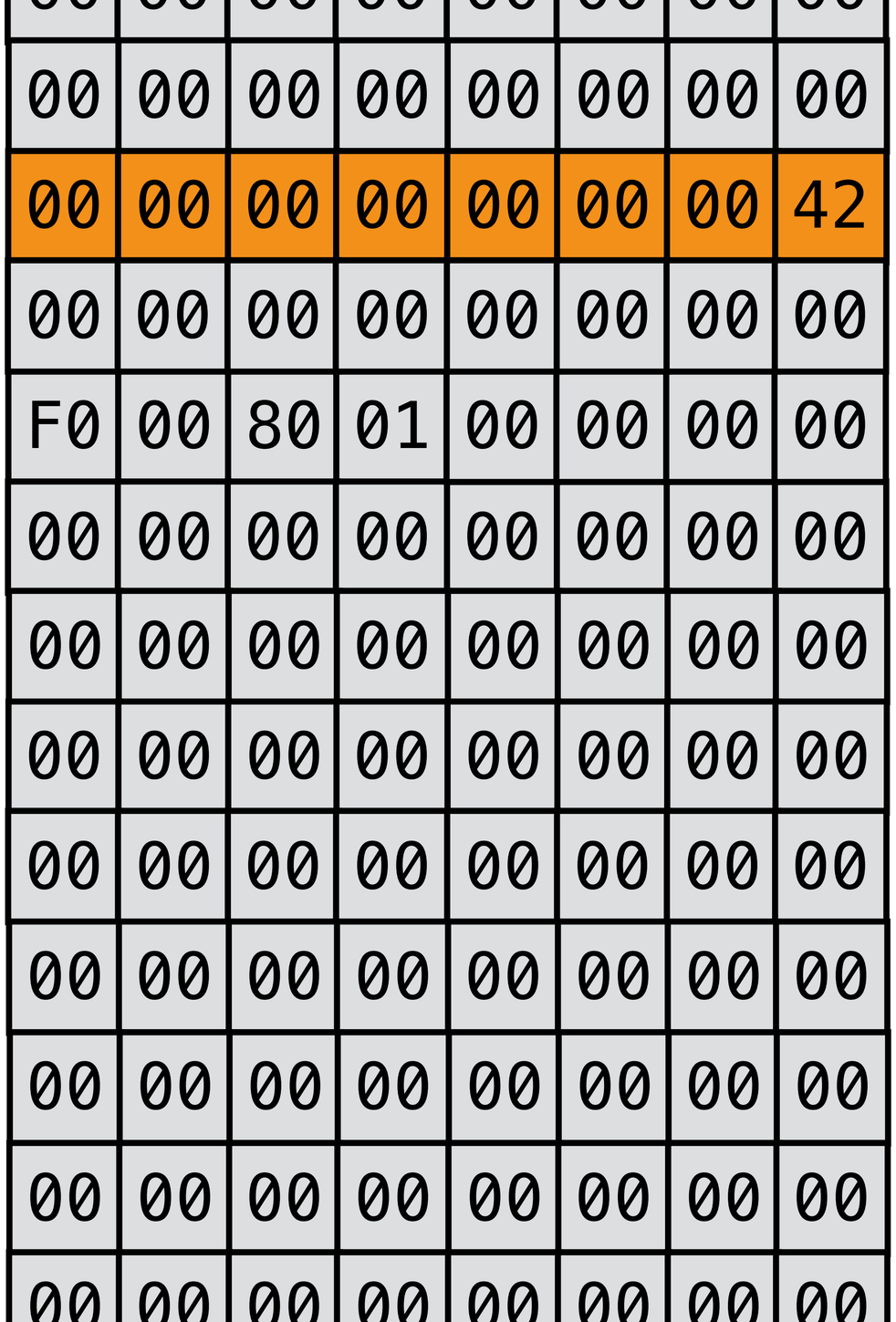
1 Object types have *alignment requirements* ([basic.fundamental], [basic.compound]) which place restrictions on the addresses at which an object of that type may be allocated. An *alignment* is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type; stricter alignment can be requested using the alignment specifier.

2 A *fundamental alignment* is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts, which is equal to `alignof(std::max_align_t)` ([support.types]). The alignment required for a type might be different when it is used as the type of a complete object and when it is used as the type of a subobject. [*Example:*

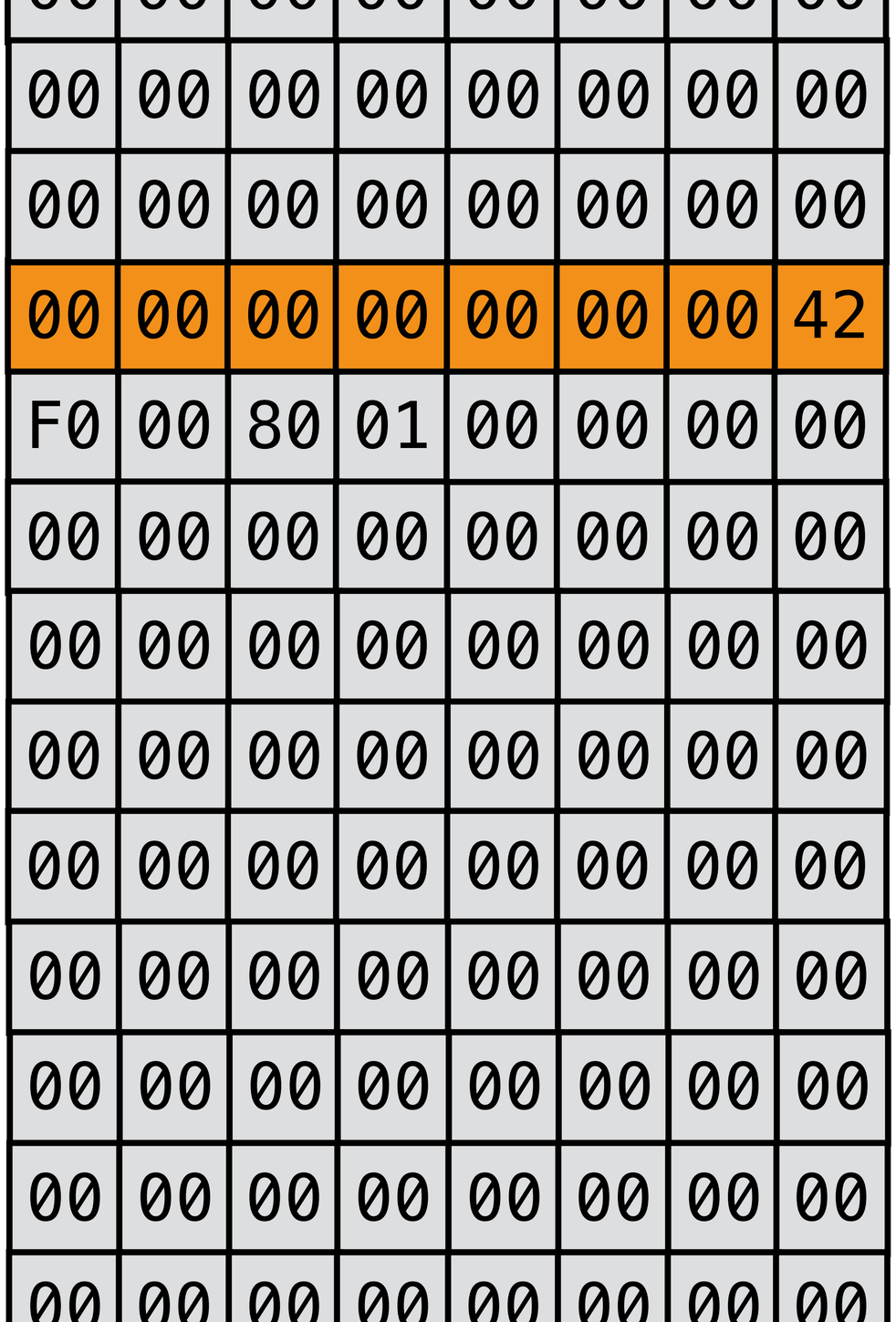
```
struct B { long double d; };
struct D : virtual B { char c; };
```

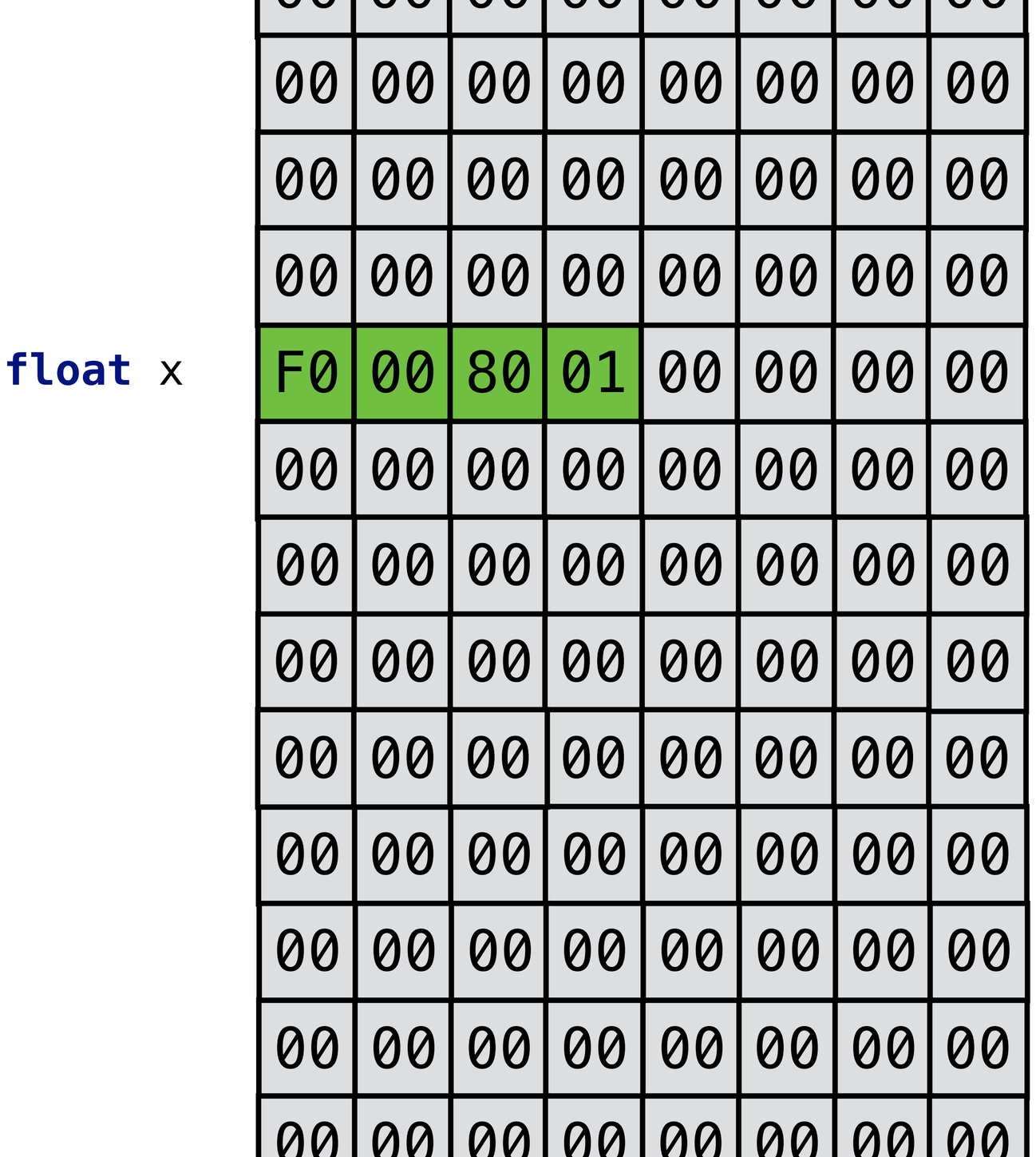
00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00
F0	00	80	01	00	00	00	00
00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00

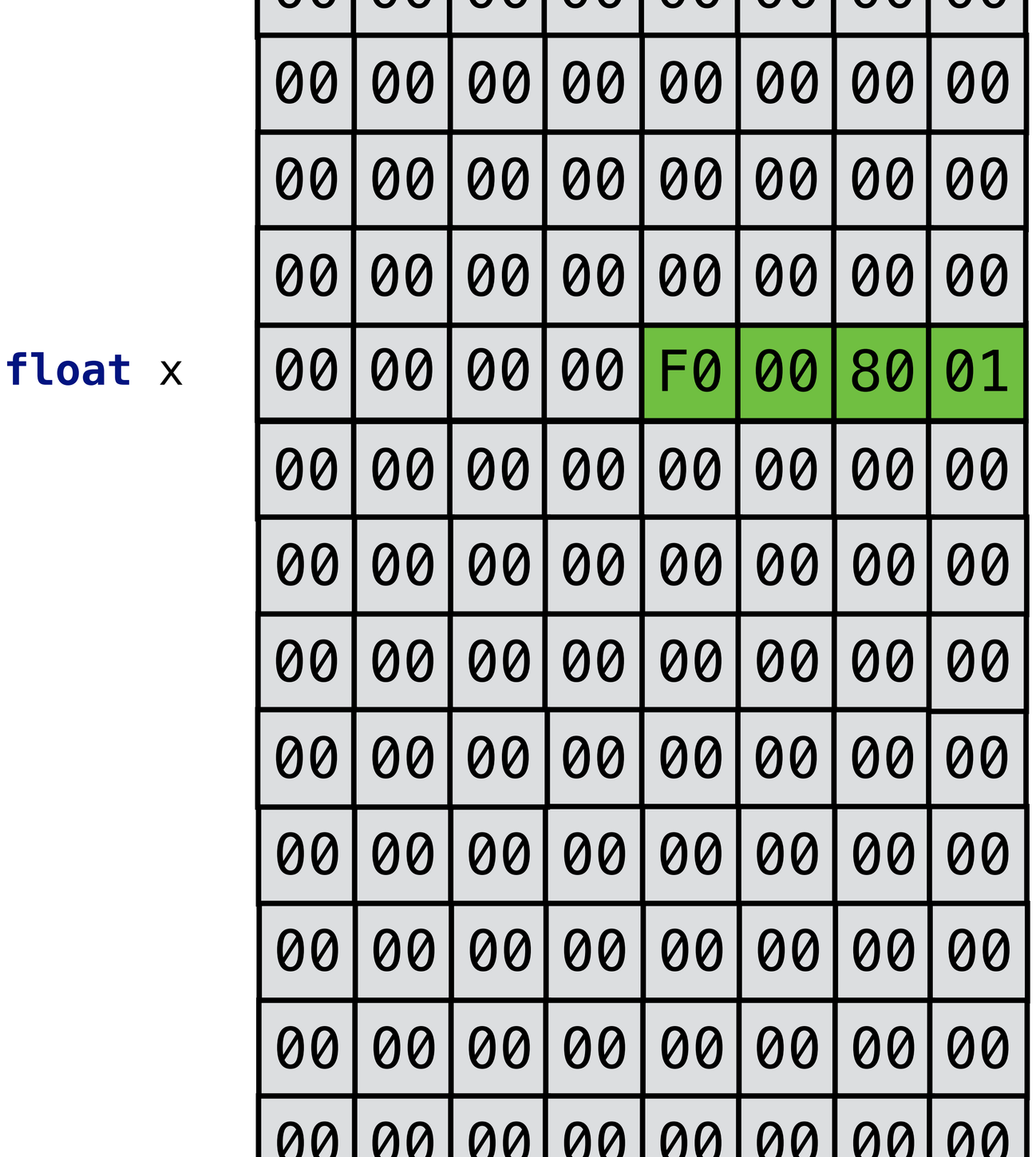
```
size_t i;
```

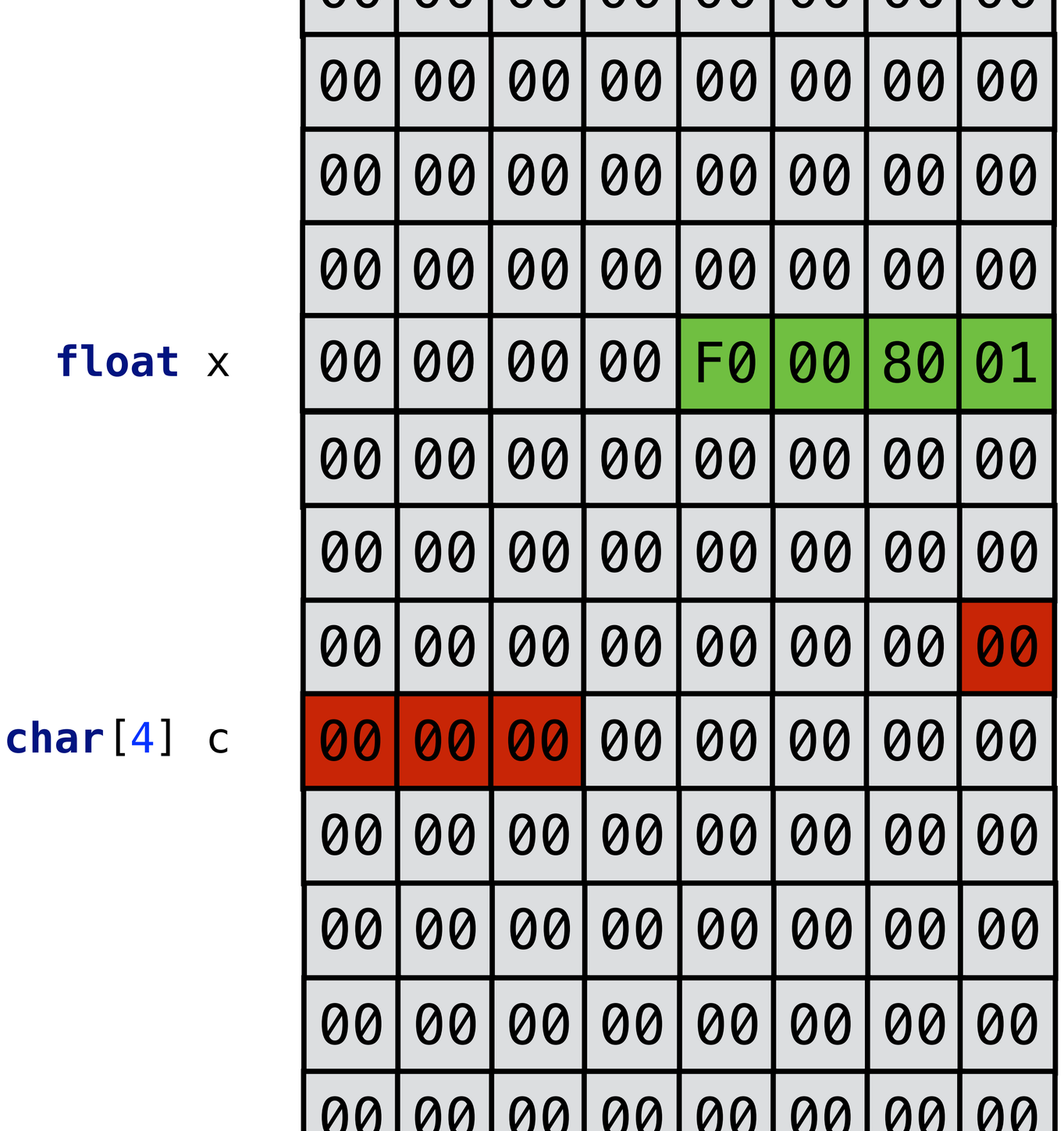

00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 42
00 00 00 00 00 00 00 00
F0 00 80 01 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00

```
size_t i;
```


00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 42
F0 00 80 01 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00


float x
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
F0 00 80 01 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00


float x
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 F0 00 80 01
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00


00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
float x
00 00 00 00 F0 00 80 01
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
char[4] c
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00

```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    char storage[sizeof(X)];
    X* p = new(storage) X;  // Undefined behaviour :(
    p->a = 1;
    p->b = 2;
    return p;
}
```

```cpp
struct X
{
    int a;
    int b;
};

X* make_x()
{
    std::aligned_storage_t<sizeof(X), alignof(X)> storage;
    X* p = new(&storage) X;  // OK :)
    p->a = 1;
    p->b = 2;
    return p;
}
```

Creating char arrays for placement-newing objects into them:

on the stack – std::aligned_storage
on the heap – std::aligned_alloc

```
float Q_rsqrt( float number )
{
    int i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;

    i  = * ( int * ) &y;
    i  = 0x5f3759df - ( i >> 1 );

    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

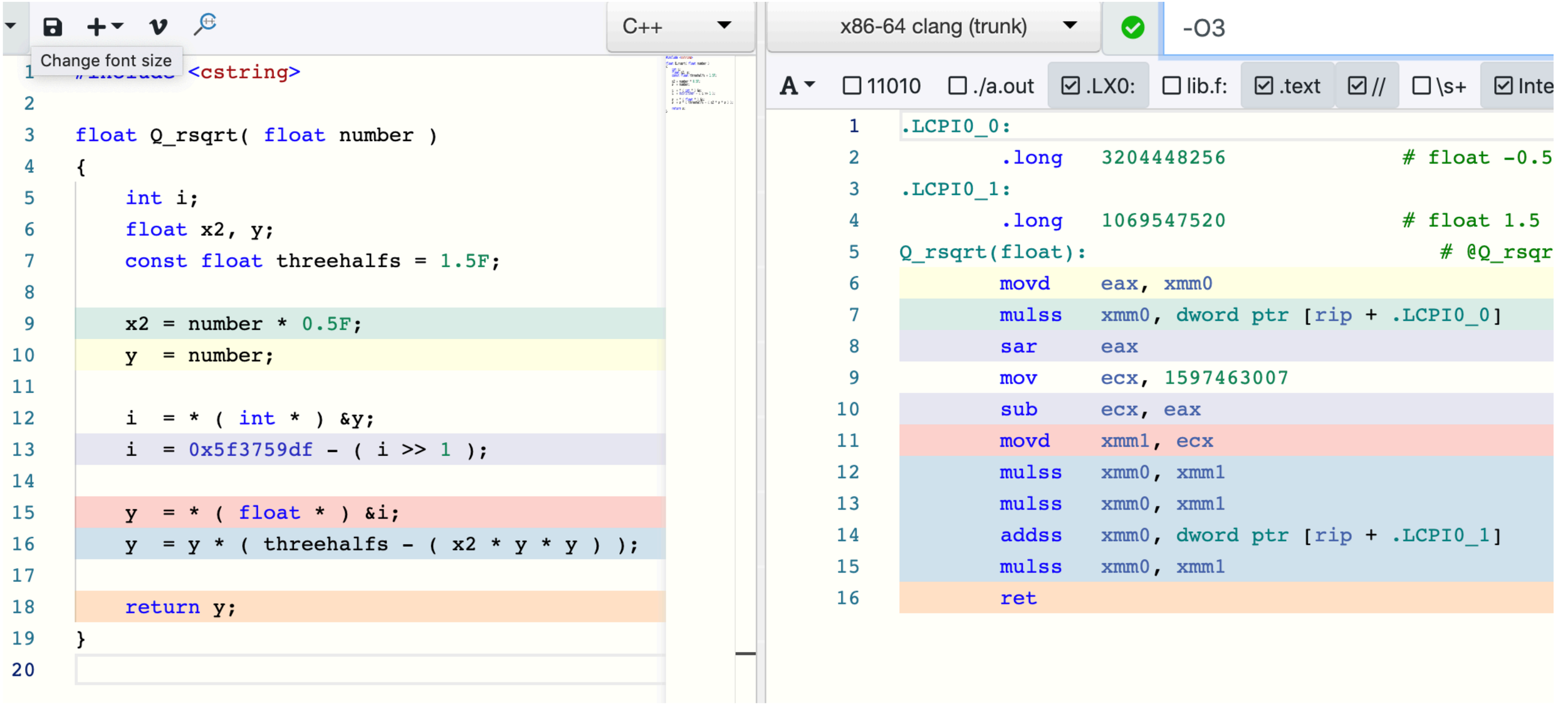
C++17

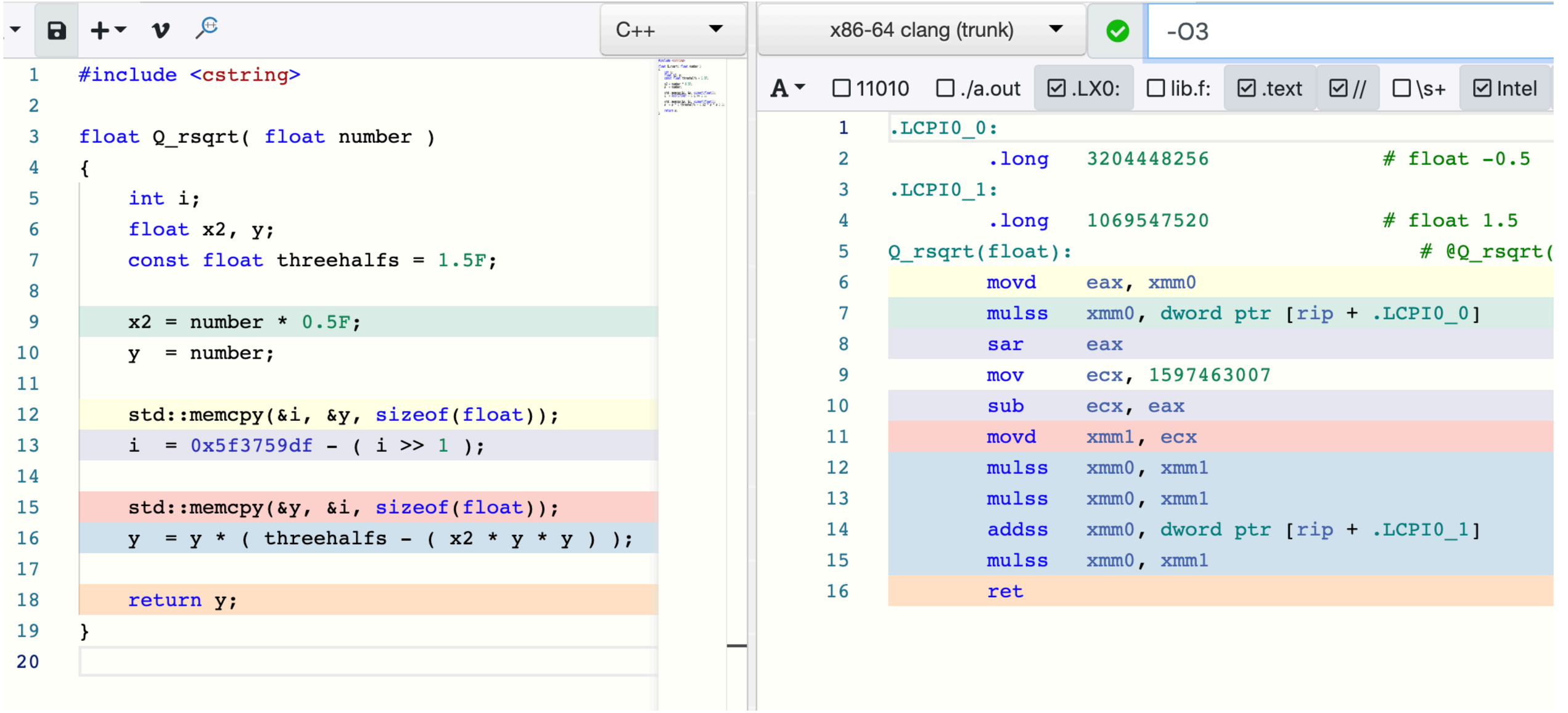
C++17

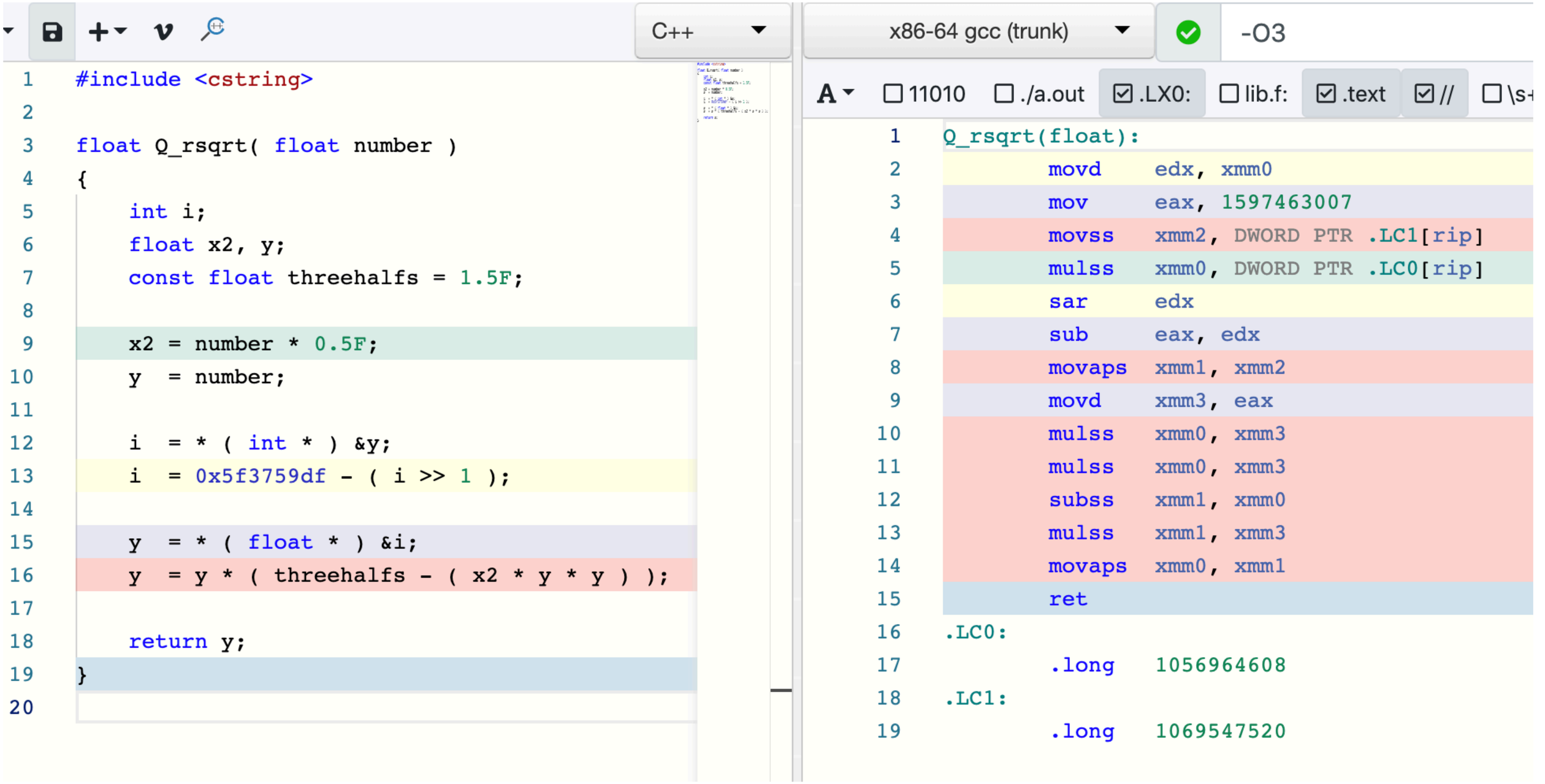
```
std::memcpy
```

```
float Q_rsqrt( float number )
{
    int i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;

    i  = * ( int * ) &y;
    i  = 0x5f3759df - ( i >> 1 );

    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

```
float Q_rsqrt( float number )
{
    int i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;

    std::memcpy(&i, &y, sizeof(float));
    i  = 0x5f3759df - ( i >> 1 );

    std::memcpy(&y, &i, sizeof(float));
    y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```


Change font size
C++
x86-64 clang (trunk)
-O3
11010
./a.out
.LX0:
lib.f:
.text
//
\s+
Inte
#include <cstring>
float Q_rsqrt(float number)
{
int i;
float x2, y;
const float threehalfs = 1.5F;
x2 = number * 0.5F;
y = number;
i = * (int *) &y;
i = 0x5f3759df - (i >> 1);
y = * (float *) &i;
y = y * (threehalfs - (x2 * y * y));
return y;
}
.LCPI0_0:
.long 3204448256 # float -0.5
.LCPI0_1:
.long 1069547520 # float 1.5
Q_rsqrt(float): # @Q_rsqr
movd eax, xmm0
mulss xmm0, dword ptr [rip + .LCPI0_0]
sar eax
mov ecx, 1597463007
sub ecx, eax
movd xmm1, ecx
mulss xmm0, xmm1
mulss xmm0, xmm1
addss xmm0, dword ptr [rip + .LCPI0_1]
mulss xmm0, xmm1
ret


C++
x86-64 clang (trunk)
-O3
11010
./a.out
.LX0:
lib.f:
.text
//
\s+
Intel
#include <cstring>
float Q_rsqrt(float number)
{
int i;
float x2, y;
const float threehalfs = 1.5F;
x2 = number * 0.5F;
y = number;
std::memcpy(&i, &y, sizeof(float));
i = 0x5f3759df - (i >> 1);
std::memcpy(&y, &i, sizeof(float));
y = y * (threehalfs - (x2 * y * y));
return y;
}
.LCPI0_0:
.long 3204448256 # float -0.5
.LCPI0_1:
.long 1069547520 # float 1.5
Q_rsqrt(float):
movd eax, xmm0
mulss xmm0, dword ptr [rip + .LCPI0_0]
sar eax
mov ecx, 1597463007
sub ecx, eax
movd xmm1, ecx
mulss xmm0, xmm1
mulss xmm0, xmm1
addss xmm0, dword ptr [rip + .LCPI0_1]
mulss xmm0, xmm1
ret


C++
x86-64 gcc (trunk)
-O3
#include <cstring>
float Q_rsqrt(float number)
{
int i;
float x2, y;
const float threehalfs = 1.5F;
x2 = number * 0.5F;
y = number;
i = * (int *) &y;
i = 0x5f3759df - (i >> 1);
y = * (float *) &i;
y = y * (threehalfs - (x2 * y * y));
return y;
}
Q_rsqrt(float):
movd edx, xmm0
mov eax, 1597463007
movss xmm2, DWORD PTR .LC1[rip]
mulss xmm0, DWORD PTR .LC0[rip]
sar edx
sub eax, edx
movaps xmm1, xmm2
movd xmm3, eax
mulss xmm0, xmm3
mulss xmm0, xmm3
subss xmm1, xmm0
mulss xmm1, xmm3
movaps xmm0, xmm1
ret
.LC0:
.long 1056964608
.LC1:
.long 1069547520

C++

```
#include <cstring>

float Q_rsqrt( float number )
{
    int i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;

    std::memcpy(&i, &y, sizeof(float));
    i  = 0x5f3759df - ( i >> 1 );

    std::memcpy(&y, &i, sizeof(float));
    y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

x86-64 gcc (trunk) -O3

A 11010 ./a.out .LX0: lib.f: .text // \s+

```
Q_rsqrt(float):
        movd    edx, xmm0
        mov     eax, 1597463007
        movss   xmm2, DWORD PTR .LC1[rip]
        mulss   xmm0, DWORD PTR .LC0[rip]
        sar     edx
        sub     eax, edx
        movaps  xmm1, xmm2
        movd    xmm3, eax
        mulss   xmm0, xmm3
        mulss   xmm0, xmm3
        subss   xmm1, xmm0
        mulss   xmm1, xmm3
        movaps  xmm0, xmm1
        ret
.LC0:
        .long   1056964608
.LC1:
        .long   1069547520
```

C++

```
1   #include <cstring>
2
3   float Q_rsqrt( float number )
4   {
5       int i;
6       float x2, y;
7       const float threehalfs = 1.5F;
8
9       x2 = number * 0.5F;
10      y  = number;
11
12      i  = * ( int * ) &y;
13      i  = 0x5f3759df - ( i >> 1 );
14
15      y  = * ( float * ) &i;
16      y  = y * ( threehalfs - ( x2 * y * y ) );
17
18      return y;
19  }
20
```

x64 msvc v19.22 /O2

A | 11010 | ./a.out | .LX0: | lib.f: | .text | // | \s+

```
1   __real@3fc00000 DD 03fc00000r          ; 1.5
2   __real@3f000000 DD 03f000000r          ; 0.5
3
4   i$ = 8
5   y$ = 8
6   number$ = 8
7   float Q_rsqrt(float) PROC
8           movaps  xmm2, xmm0
9           mov     eax, 1597463007
10          movss   xmm0, DWORD PTR __real@3fc00000
11          movss   DWORD PTR i$[rsp], xmm2
12          mov     ecx, DWORD PTR i$[rsp]
13          mulss   xmm2, DWORD PTR __real@3f000000
14          sar     ecx, 1
15          sub     eax, ecx
16          mov     DWORD PTR y$[rsp], eax
17          movss   xmm1, DWORD PTR y$[rsp]
18          mulss   xmm2, xmm1
19          mulss   xmm2, xmm1
20          subss   xmm0, xmm2
21          mulss   xmm0, xmm1
22          ret     0
23  float Q_rsqrt(float) ENDP
```

C++

```cpp
#include <cstring>

float Q_rsqrt( float number )
{
    int i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;

    std::memcpy(&i, &y, sizeof(float));
    i  = 0x5f3759df - ( i >> 1 );

    std::memcpy(&y, &i, sizeof(float));
    y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

x64 msvc v19.22 /O2

A | 11010 | ./a.out | .LX0: | lib.f: | .text | // | \s+ | Int

```
__real@3fc00000 DD 03fc00000r                   ; 1.5
__real@3f000000 DD 03f000000r                   ; 0.5

i$ = 8
y$ = 8
number$ = 8
float Q_rsqrt(float) PROC
        movaps  xmm2, xmm0
        mov     eax, 1597463007
        movss   xmm0, DWORD PTR __real@3fc00000
        movss   DWORD PTR i$[rsp], xmm2
        mov     ecx, DWORD PTR i$[rsp]
        mulss   xmm2, DWORD PTR __real@3f000000
        sar     ecx, 1
        sub     eax, ecx
        mov     DWORD PTR y$[rsp], eax
        movss   xmm1, DWORD PTR y$[rsp]
        mulss   xmm2, xmm1
        mulss   xmm2, xmm1
        subss   xmm0, xmm2
        mulss   xmm0, xmm1
        ret     0
float Q_rsqrt(float) ENDP
```

```cpp
int main()
{
    T t = {};
    U u;

    std::memcpy(&u, &t, sizeof(U));
}
```

```cpp
int main()
{
    T t = {};
    U u;

    static_assert(sizeof(T) == sizeof(U));
    std::memcpy(&u, &t, sizeof(U));
}
```

```cpp
int main()
{
    T t = {};
    U u;

    static_assert(sizeof(T) == sizeof(U));
    static_assert(std::is_trivially_copyable_v<T>);
    static_assert(std::is_trivially_copyable_v<U>);
    std::memcpy(&u, &t, sizeof(U));
}
```

```
template <typename To,
          typename From,
          typename = std::enable_if_t<
              (sizeof(To) == sizeof(From)) &&
              std::is_trivially_copyable_v<From> &&
              std::is_trivially_copyable_v<To>>>
To bit_cast(const From &src) noexcept
{
    To dst;
    std::memcpy(&dst, &src, sizeof(To));
    return dst;
}
```

C++20

```
std::bit_cast
```

```
float Q_rsqrt( float y )
{
    const auto threehalfs = 1.5f;
    const auto x2 = y * 0.5f;

    auto i = std::bit_cast<int>(y);
    i  = 0x5f3759df - ( i >> 1 );

    y = std::bit_cast<float>(i);
    y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

```
constexpr float Q_rsqrt( float y )
{
    constexpr auto threehalfs = 1.5f;
    constexpr auto x2 = y * 0.5f;

    auto i = std::bit_cast<int>(y);
    i  = 0x5f3759df - ( i >> 1 );

    y = std::bit_cast<float>(i);
    y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

26.5.3 Function template `bit_cast` [bit.cast]

```
template<class To, class From>
  constexpr To bit_cast(const From& from) noexcept;
```

1 *Returns:* An object of type `To`. Each bit of the value representation of the result is equal to the corresponding bit in the object representation of `from`. Padding bits of the `To` object are unspecified. If there is no value of type `To` corresponding to the value representation produced, the behavior is undefined. If there are multiple such values, which value is produced is unspecified.

2 *Remarks:* This function shall not participate in overload resolution unless:

(2.1) — `sizeof(To) == sizeof(From)` is `true`;

(2.2) — `is_trivially_copyable_v<To>` is `true`; and

(2.3) — `is_trivially_copyable_v<From>` is `true`.

This function shall be `constexpr` if and only if `To`, `From`, and the types of all subobjects of `To` and `From` are types `T` such that:

26.5.3 Function template `bit_cast` [bit.cast]

```
template<class To, class From>
  constexpr To bit_cast(const From& from) noexcept;
```

1 *Returns:* An object of type `To`. Each bit of the value representation of the result is equal to the corresponding bit in the object representation of `from`. Padding bits of the `To` object are unspecified. If there is no value of type `To` corresponding to the value representation produced, the behavior is undefined. If there are multiple such values, which value is produced is unspecified.

2 *Remarks:* This function shall not participate in overload resolution unless:

(2.1) — `sizeof(To) == sizeof(From)` is `true`;

(2.2) — `is_trivially_copyable_v<To>` is `true`; and

(2.3) — `is_trivially_copyable_v<From>` is `true`.

This function shall be `constexpr` if and only if `To`, `From`, and the types of all subobjects of `To` and `From` are types `T` such that:

- aliasing rules
- object lifetime rules
- alignment rules
- rules for valid value representations

```
constexpr float Q_rsqrt( float y )
{
    constexpr auto threehalfs = 1.5f;
    constexpr auto x2 = y * 0.5f;

    auto i = std::bit_cast<int>(y);
    i  = 0x5f3759df - ( i >> 1 );

    y = std::bit_cast<float>(i);
    y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

P0907R4
Signed Integers are Two's Complement

Published Proposal, 2018-10-06

This version:

http://wg21.link/P0907R4

Issue Tracking:

Inline In Spec

Author:

JF Bastien (Apple)

Audience:

CWG

Toggle Diffs:

☐ Hide deleted text

Project:

ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Source:

github.com/jfbastien/papers/blob/master/source/P0907R4.bs

26.5.3 Function template `bit_cast` [bit.cast]

```
template<class To, class From>
  constexpr To bit_cast(const From& from) noexcept;
```

1 *Returns:* An object of type `To`. Each bit of the value representation of the result is equal to the corresponding bit in the object representation of `from`. Padding bits of the `To` object are unspecified. If there is no value of type `To` corresponding to the value representation produced, the behavior is undefined. If there are multiple such values, which value is produced is unspecified.

2 *Remarks:* This function shall not participate in overload resolution unless:

(2.1) — `sizeof(To) == sizeof(From)` is `true`;

(2.2) — `is_trivially_copyable_v<To>` is `true`; and

(2.3) — `is_trivially_copyable_v<From>` is `true`.

This function shall be `constexpr` if and only if `To`, `From`, and the types of all subobjects of `To` and `From` are types `T` such that:

C++23?

float

F0 00 80 01

int

F0 00 80 01

---

float

F0 00 80 01

char*

std::byte*

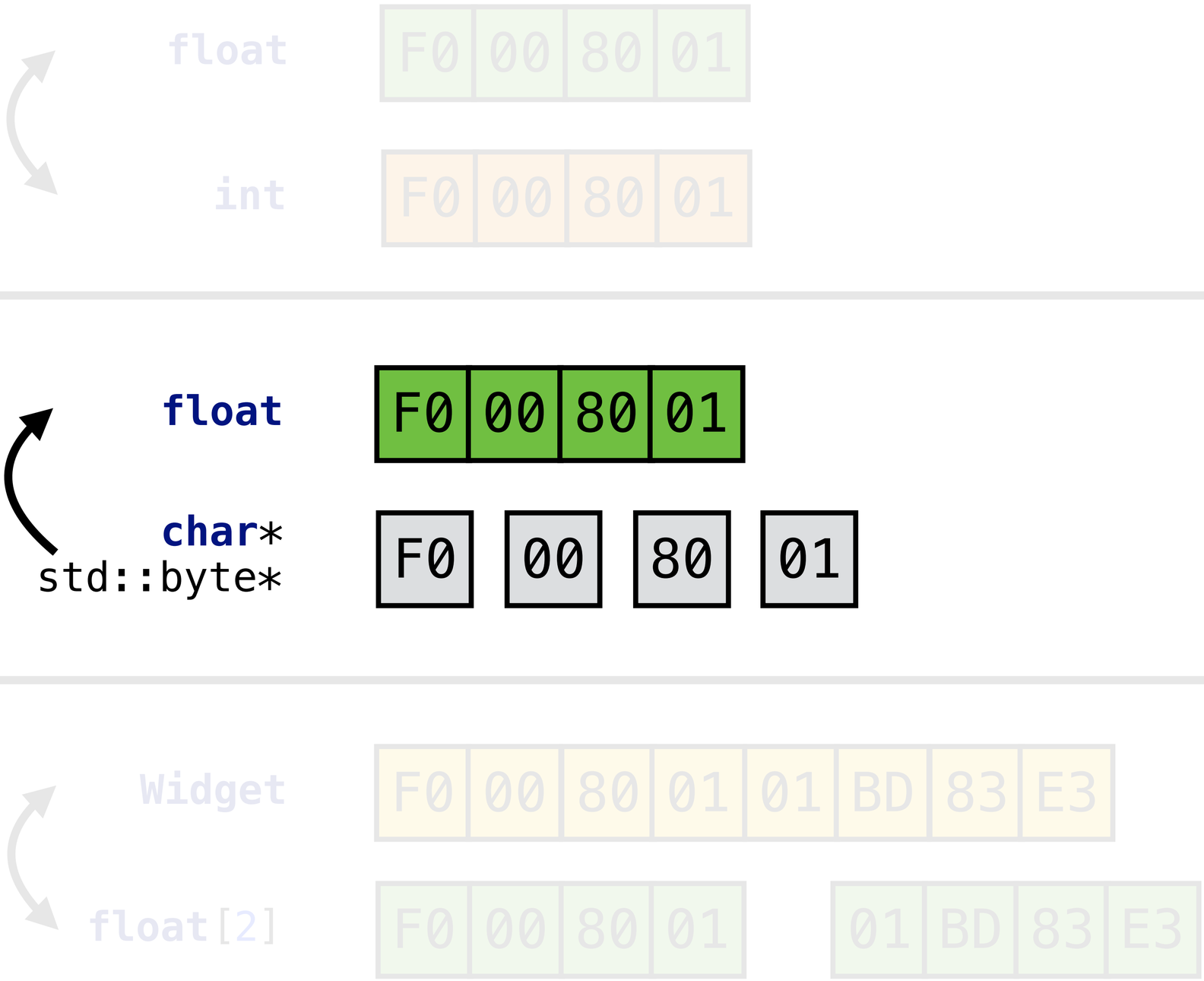
F0 00 80 01

Widget

F0 00 80 01 01 BD 83 E3

float[2]

F0 00 80 01 01 BD 83 E3


float
F0 00 80 01
int
F0 00 80 01
float
F0 00 80 01
char*
std::byte*
F0 00 80 01
Widget
F0 00 80 01 01 BD 83 E3
float[2]
F0 00 80 01
01 BD 83 E3

```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    X* p = (X*)malloc(sizeof(struct X));   // UB: there is no object of type X here!
    p->a = 1;
    p->b = 2;
    return p;
}
```

```
void process(Stream* stream)
{
    std::unique_ptr<char[]> buffer = stream->read();

    if (buffer[0] == WIDGET)
        processWidget(reinterpret_cast<Widget*>(buffer.get()));    // UB :(
    else
        // ...
}
```

6.6.2 Object model [intro.object]

1 The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is created by a definition, by a *new-expression*, when implicitly changing the active member of a union, or when a temporary object is created ([conv.rval], [class.temporary]). An object occupies a region of storage in its period of construction ([class.cdtor]), throughout its lifetime, and in its period of destruction ([class.cdtor]). [*Note:* A function is not an object, regardless of whether or not it occupies storage in the way that objects do. — *end note*] The properties of an object are determined when the object is created. An object can have a name. An object has a storage duration which influences its lifetime. An object has a type. Some objects are polymorphic ([class.virtual]); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the *expressions* ([expr.compound]) used to access them.

P0593R5
Implicit creation of objects for low-level object manipulation

Published Proposal, 2019-10-06

This version:
http://wg21.link/p0593r5

Author:
Richard Smith (Google)

Former Author:
Ville Voutilainen

Audience:
EWG, LWG, CWG

Project:
ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Abstract

This paper proposes that objects of sufficiently trivial types be created on-demand as necessary within newly-allocated storage to give programs defined behavior.

§ 5.1. 6.6.2 Object model [intro.object]

Change in 6.6.2 [intro.object] paragraph 1:

> The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is created by a definition (6.1), by a new-expression (7.6.2.4), by an operation that implicitly creates objects (see below), when implicitly changing the active member of a union (10.4), or when a temporary object is created (7.3.4, 6.6.7). [...]

Add a new paragraph at the end of [intro.object]:

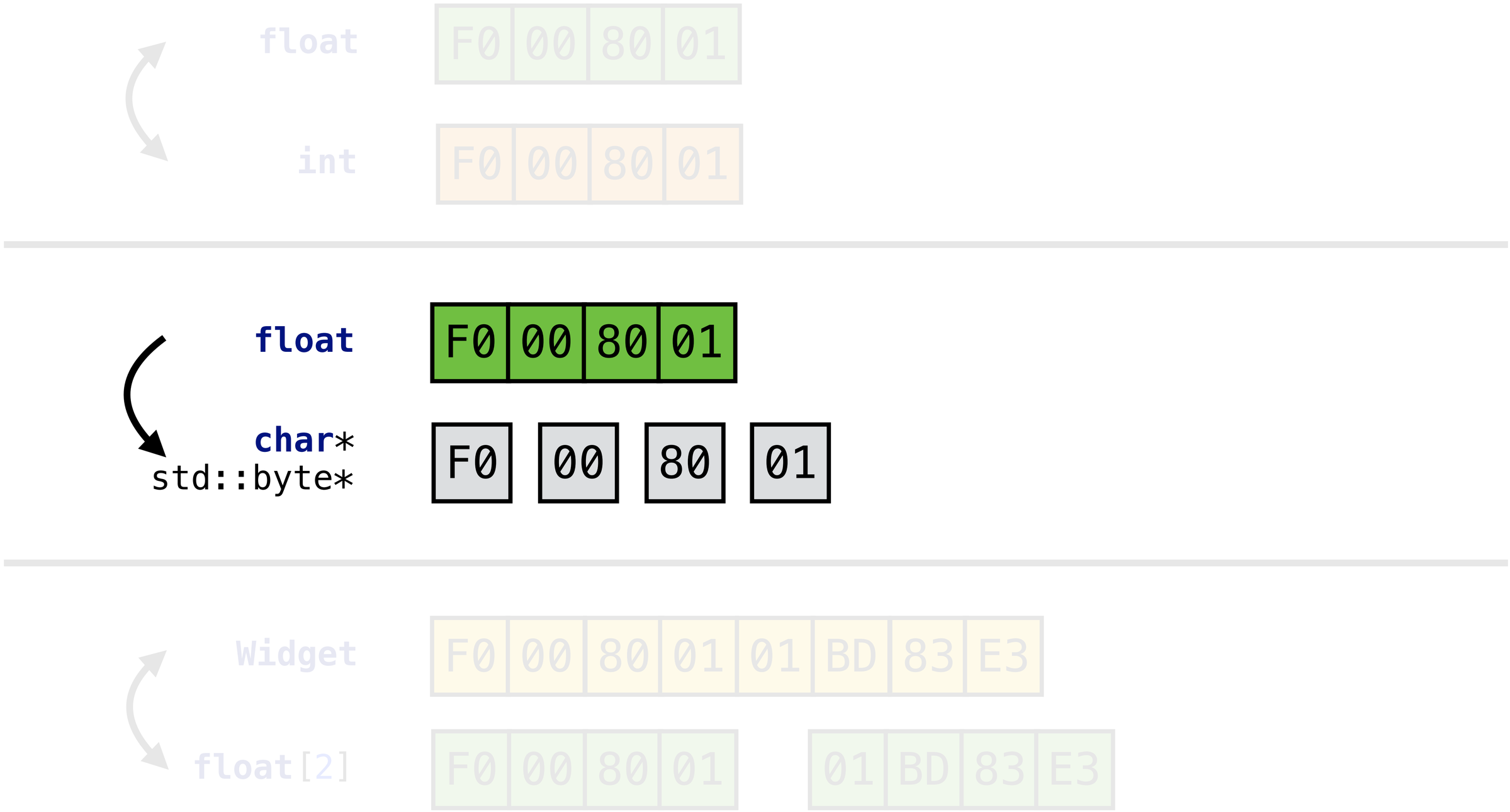
> Some operations are described as *implicitly creating objects* within a specified region of storage. For each operation that is specified as implicitly creating objects, that operation implicitly creates and starts the lifetime of zero or more objects of implicit-lifetime types (6.7 [basic.types]) in its specified region of storage if doing so would result in the program having defined behavior. If no such sets of objects would give the program defined behavior, the behavior of the program is undefined. [*Note*: Such operations do not start the lifetimes of subobjects of such objects that are not themselves of implicit-lifetime types. -*end note*]

Add another paragraph:

> An operation that begins the lifetime of an array of `char`, `unsigned char`, or `std::byte` implicitly creates objects within the region of storage occupied by the array. [Note: the array object provides storage for these objects. -- end note] Any implicit or explicit invocation of a function named `operator new` or `operator new[]` implicitly creates objects in the returned region of storage. Some functions in the C++ standard library implicitly create objects (19.10.9.2 [allocator.traits.members], 19.10.12 [c.malloc], 20.5.3 [cstring.syn], 26.5.3 [bit.cast]).

```
struct X
{
    int a;
    int b;
};

X* make_x()
{
    X* p = (X*)malloc(sizeof(struct X));   // P0593: malloc implicitly creates an X :)
    p->a = 1;
    p->b = 2;
    return p;
}
```

```
void process(Stream* stream)
{
    std::unique_ptr<char[]> buffer = stream->read();

    if (buffer[0] == WIDGET)
        processWidget(std::start_lifetime_as<Widget>(buffer.get()));  // new in P0593
    else
        // ...
}
```


float
F0 00 80 01
int
F0 00 80 01
float
F0 00 80 01
char*
std::byte*
F0 00 80 01
Widget
F0 00 80 01 01 BD 83 E3
float[2]
F0 00 80 01
01 BD 83 E3

```
void printBitRepresentation(float f)
{
    // implementation ???
}
```

```
void printBitRepresentation(float f)
{
    auto* buf = reinterpret_cast<unsigned char*>(&f);
    for (int i = 0; i < sizeof(float); ++i)
        std::cout << buf[i];
}
```

```
void printBitRepresentation(float f)
{
    auto* buf = reinterpret_cast<unsigned char*>(&f);
    for (int i = 0; i < sizeof(float); ++i)
        std::cout << buf[i];
}
```

```
void printBitRepresentation(float f)
{
    auto* buf = reinterpret_cast<unsigned char*>(&f);
    for (int i = 0; i < sizeof(float); ++i)
        std::cout << buf[i];
}
```

```
void printBitRepresentation(float f)
{
    auto* buf = reinterpret_cast<unsigned char*>(&f);
    for (int i = 0; i < sizeof(float); ++i)
        std::cout << buf[i];    // UB :(
}
```

```
void printBitRepresentation(float f)
{
    unsigned char buf[sizeof(float)];
    std::memcpy(&buf, &f, sizeof(float));
    for (auto c : buf)
        std::cout << c;
}
```

```cpp
void printBitRepresentation(float f)
{
    auto* buf = reinterpret_cast<unsigned char*>(&f);
    for (int i = 0; i < sizeof(float); ++i)
        std::cout << buf[i];    // UB :(
}
```

Accessing Object Representations

Document #:	P1839R1
Date:	2019-09-28
Project:	Programming Language C++ Core Working Group

Reply-to:	Krystian Stasiowski <sdkrystian@gmail.com>

1 Abstract

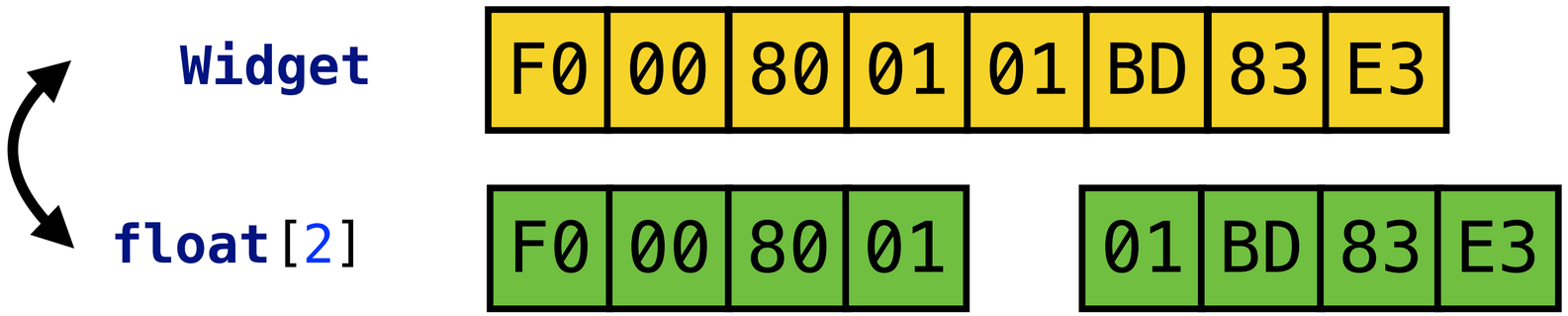
Allow access to the object representation of an object.

3 Motivation

This proposal does not intend to introduce anything new, but rather to standardize a common existing practice. Accessing the underlying bytes of an object has been a long-standing practice in C and C++ alike, but in C++, doing so is typically undefined behavior. With current wording, it is impossible to obtain a pointer to an element of the object representation, with an expression such as `reinterpret_cast<char*>(&a)` typically yielding a pointer to the original object, with only the type of the expression being changed. This does not represent the intent of CWG, as exemplified by [CWG1314] in which it is stated that access to the object representation is intended to be well defined.

float
F0 00 80 01
int
F0 00 80 01

float
F0 00 80 01
char*
std::byte*
F0 00 80 01


Widget
F0 00 80 01 01 BD 83 E3
float[2]
F0 00 80 01
01 BD 83 E3

26.4 Complex numbers [complex.numbers]

1 The header `<complex>` defines a class template, and numerous functions for representing and manipulating complex numbers.

2 The effect of instantiating the template `complex` for any type other than `float`, `double`, or `long double` is unspecified. The specializations `complex<float>`, `complex<double>`, and `complex<long double>` are literal types.

3 If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

4 If `z` is an lvalue of type *cv* `complex<T>` then:

(4.1) — the expression `reinterpret_cast<cv T(&)[2]>(z)` shall be well-formed,

(4.2) — `reinterpret_cast<cv T(&)[2]>(z)[0]` shall designate the real part of `z`, and

(4.3) — `reinterpret_cast<cv T(&)[2]>(z)[1]` shall designate the imaginary part of `z`.

Moreover, if `a` is an expression of type *cv* `complex<T>*` and the expression `a[i]` is well-defined for an integer expression `i`, then:

(4.4) — `reinterpret_cast<cv T*>(a)[2*i]` shall designate the real part of `a[i]`, and

(4.5) — `reinterpret_cast<cv T*>(a)[2*i + 1]` shall designate the imaginary part of `a[i]`.

```
struct Widget
{
    int i = 42;
    float f = 0;
};

int main()
{
    Widget w;
    int i = *reinterpret_cast<int*>(&w);
    return i;
}
```

```
struct Widget
{
    int i = 42;
    float f = 0;
};

int main()
{
    Widget w;
    int i = *reinterpret_cast<int*>(&w);
    return i;
}
```

```
struct Widget
{
    int i = 42;
    float f = 0;
};

int main()
{
    Widget w;
    int i = *reinterpret_cast<int*>(&w);  // OK: Widget and int are
    return i;                             // pointer-interconvertible
}
```

4 Two objects *a* and *b* are *pointer-interconvertible* if:

(4.1) — they are the same object, or

(4.2) — one is a union object and the other is a non-static data member of that object ([class.union]), or

(4.3) — one is a standard-layout class object and the other is the first non-static data member of that object, or, if the object has no non-static data members, any base class subobject of that object ([class.mem]), or

(4.4) — there exists an object *c* such that *a* and *c* are pointer-interconvertible, and *c* and *b* are pointer-interconvertible.

If two objects are pointer-interconvertible, then they have the same address, and it is possible to obtain a pointer to one from a pointer to the other via a `reinterpret_cast`. [*Note:* An array object and its first element are not pointer-interconvertible, even though they have the same address. — *end note*]

5 A pointer to *cv*-qualified ([basic.type.qualifier]) or *cv*-unqualified `void` can be used to point to objects of unknown type. Such a pointer shall be able to hold any object pointer. An object of type *cv* `void*` shall have the same representation and alignment requirements as *cv* `char*`.

```cpp
template <typename T>
struct complex
{
    T data[2];
};

int main()
{
    complex<float> c = {1.0, 0.0};
    float real = reinterpret_cast<float(&)[2]>(c)[0];   // OK
}
```

```cpp
template <typename T>
struct complex
{
    T real, imag;
};

int main()
{
    complex<float> c = {1.0, 0.0};
    float real = reinterpret_cast<float(&)[2]>(c)[0];   // UB, unless you are
                                                        // the compiler vendor
}
```

26.4 Complex numbers [complex.numbers]

1 The header `<complex>` defines a class template, and numerous functions for representing and manipulating complex numbers.

2 The effect of instantiating the template `complex` for any type other than `float`, `double`, or `long double` is unspecified. The specializations `complex<float>`, `complex<double>`, and `complex<long double>` are literal types.

3 If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

4 If `z` is an lvalue of type *cv* `complex<T>` then:

(4.1) — the expression `reinterpret_cast<cv T(&)[2]>(z)` shall be well-formed,

(4.2) — `reinterpret_cast<cv T(&)[2]>(z)[0]` shall designate the real part of `z`, and

(4.3) — `reinterpret_cast<cv T(&)[2]>(z)[1]` shall designate the imaginary part of `z`.

Moreover, if `a` is an expression of type *cv* `complex<T>*` and the expression `a[i]` is well-defined for an integer expression `i`, then:

(4.4) — `reinterpret_cast<cv T*>(a)[2*i]` shall designate the real part of `a[i]`, and

(4.5) — `reinterpret_cast<cv T*>(a)[2*i + 1]` shall designate the imaginary part of `a[i]`.

```
struct __m128
{
    // four floats as a SIMD pack
};

void processBlock (float* inBlock)
{
    auto* simdBlock = reinterpret_cast<__m128*> (inBlock);
    processVectorised (simdBlock);
}
```

```
struct __m128
{
    // four floats as a SIMD pack
};

void processBlock (float* inBlock)
{
    auto* simdBlock = reinterpret_cast<__m128*> (inBlock);  // UB
    processVectorised (simdBlock);
}
```

Interconvertible object representations

Timur Doumler (papers@timur.audio)

Document #: P1912R0
Date: 2019-10-06
Project: Programming Language C++
Audience: Evolution Working Group

Abstract

We propose a new specifier declaring that a given type `T1` has an object representation compatible with some other type `T2`, in cases where the implementation can verify this at compile time. This allows to interconvert pointers and references to those types without invoking undefined behaviour. This facility plugs a very unfortunate hole in the current C++ type system.

As a side benefit, our proposed facility regularises the unusual interconvertibility properties of `std::complex`, such that it is no longer a "magic" type unimplementable without special compiler support.

```
struct __m128
{
    // four floats as a SIMD pack
};

void processBlock (float* inBlock)
{
    auto* simdBlock = reinterpret_cast<__m128*> (inBlock);  // UB
    processVectorised (simdBlock);
}
```

```
struct __m128 layoutas(float[4])  // possible new attribute-like thing here...
{
    // four floats as a SIMD pack
};

void processBlock (float* inBlock)
{
    auto* simdBlock = reinterpret_cast<__m128*> (inBlock);  // ...to make this OK!
    processVectorised (simdBlock);
}
```

Takeaways

Takeaways

- Don’t use C style casts
- Don’t use reinterpret_cast
- Don’t use unions

Takeaways

- Don’t use C style casts
- Don’t use reinterpret_cast
- Don’t use unions

Because:

- Aliasing rules
- object lifetime rules
- alignment rules
- rules for valid value representations

Takeaways

- Don’t use C style casts
- Don’t use reinterpret_cast
- Don’t use unions

C++17

- placement new
- memcpy

Because:

- Aliasing rules
- object lifetime rules
- alignment rules
- rules for valid value representations

Takeaways

- Don’t use C style casts
- Don’t use reinterpret_cast
- Don’t use unions

Because:

- Aliasing rules
- object lifetime rules
- alignment rules
- rules for valid value representations

C++17

- placement new
- memcpy

C++20

- bit_cast

Takeaways

- Don’t use C style casts
- Don’t use reinterpret_cast
- Don’t use unions

Because:

- Aliasing rules
- object lifetime rules
- alignment rules
- rules for valid value representations

C++17

- placement new
- memcpy

C++20

- bit_cast

C++23

- more tools and fixes coming!

Type punning in modern C++

version 1.1

Timur Doumler

@timur_audio

C++ Russia
30 October 2019

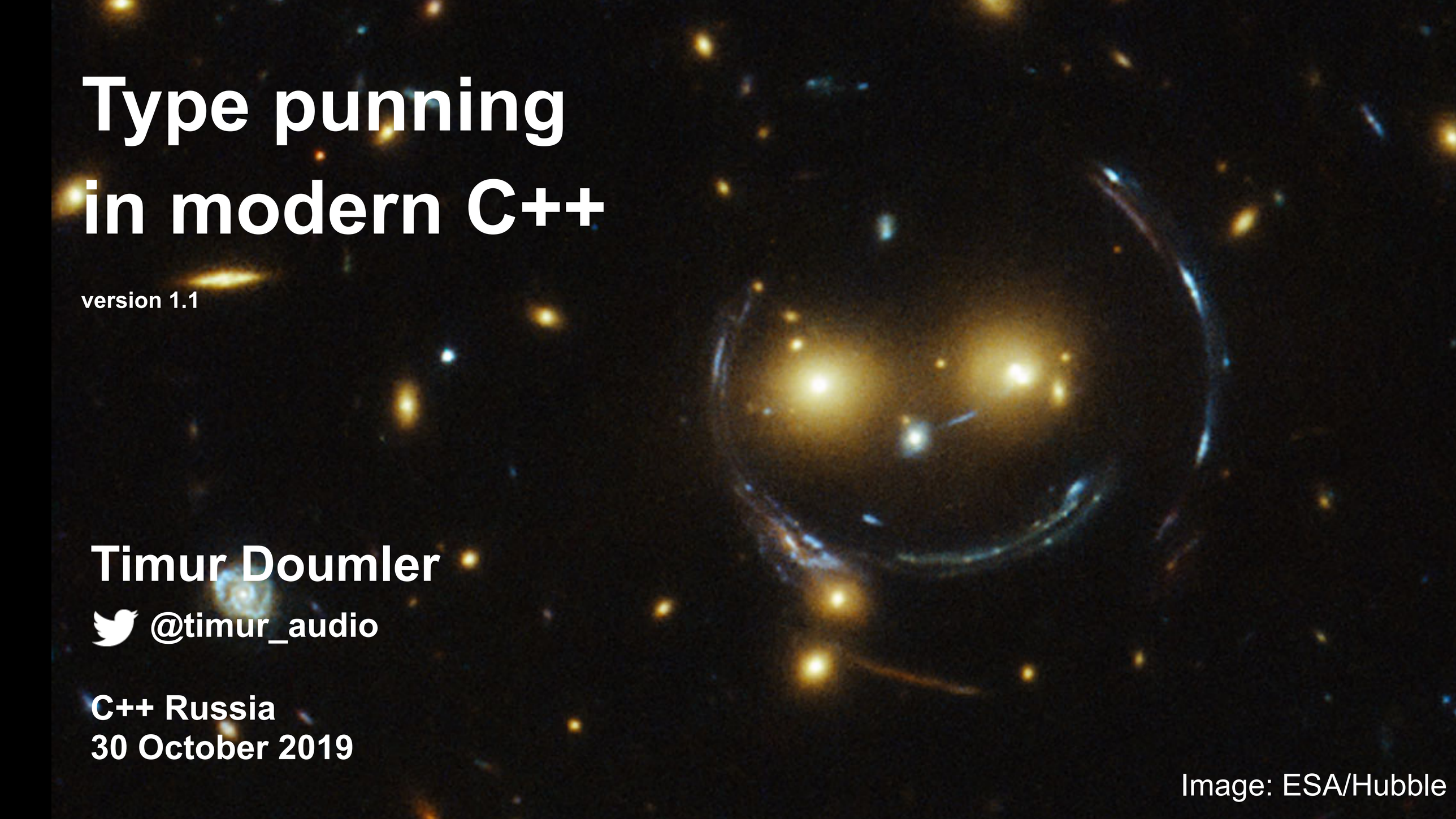


Image: ESA/Hubble