Introduction to C++

F. Giacomini

INFN-CNAF

June 2016

What is C++

C++ is a programming language that is:

- strongly and statically typed
- general-purpose
- multi-paradigm
- efficient (*"you don't pay for what you don't use"*)
- standard

What is C++

C++ is a programming language that is:

- strongly and statically typed
- general-purpose
- multi-paradigm
- efficient (*"you don't pay for what you don't use"*)
- standard

What is C++

C++ is a programming language that is:

- strongly and statically typed
- general-purpose
- multi-paradigm
- efficient (*"you don't pay for what you don't use"*)
- standard

What is C++

C++ is a programming language that is:

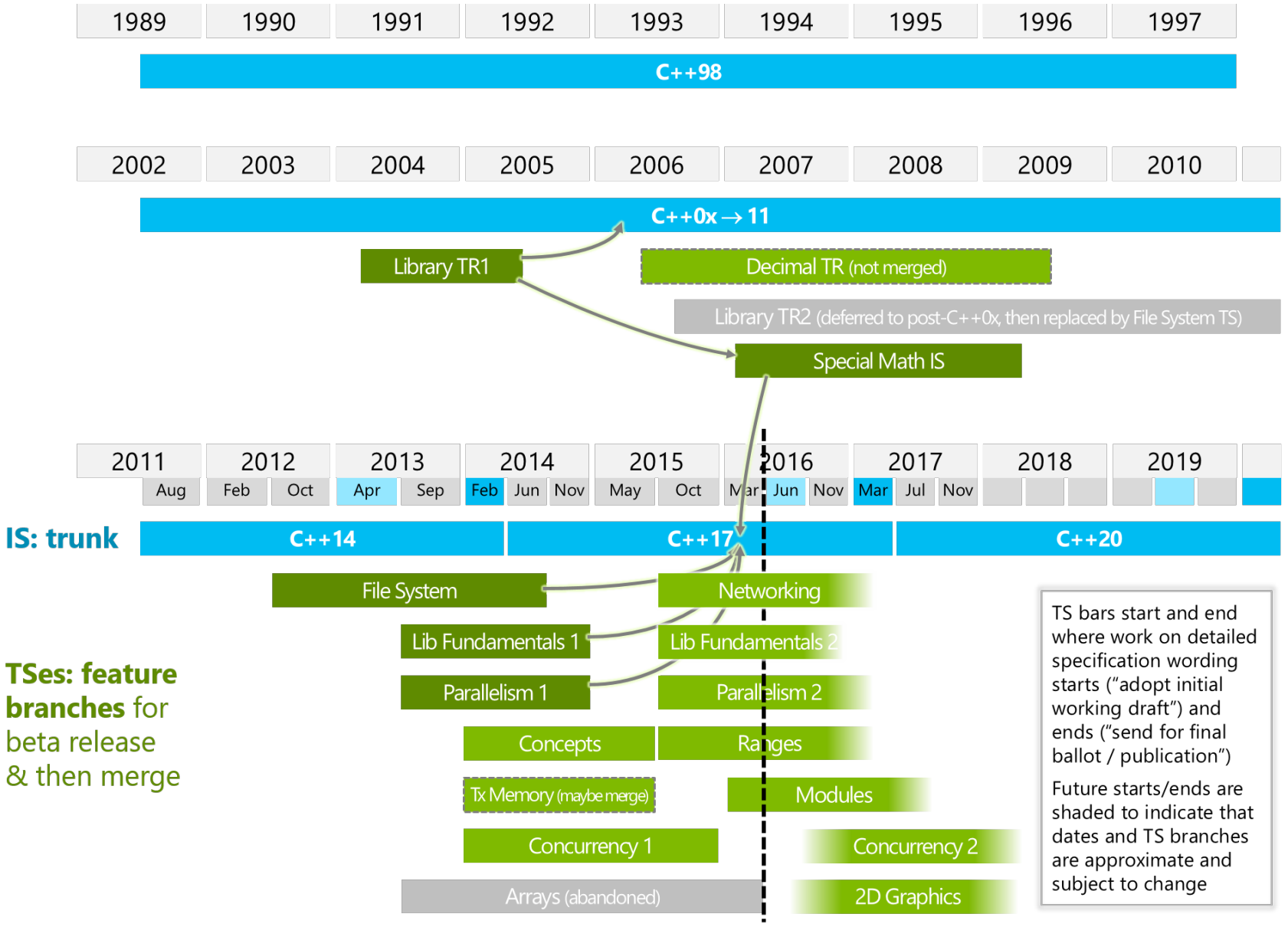
- strongly and statically typed
- general-purpose
- multi-paradigm
- efficient (*"you don't pay for what you don't use"*)
- standard

What is C++

C++ is a programming language that is:

- strongly and statically typed
- general-purpose
- multi-paradigm
- efficient (*"you don't pay for what you don't use"*)
- standard

C++ timeline


1989
1990
1991
1992
1993
1994
1995
1996
1997
C++98
2002
2003
2004
2005
2006
2007
2008
2009
2010
C++0x → 11
Library TR1
Decimal TR (not merged)
Library TR2 (deferred to post-C++0x, then replaced by File System TS)
Special Math IS
2011
Aug
2012
Feb
Oct
2013
Apr
Sep
2014
Feb
Jun
Nov
2015
May
Oct
2016
Mar
Jun
Nov
2017
Mar
Jul
Nov
2018
2019
IS: trunk
C++14
C++17
C++20
File System
Networking
Lib Fundamentals 1
Lib Fundamentals 2
Parallelism 1
Parallelism 2
Concepts
Ranges
Tx Memory (maybe merge)
Modules
Concurrency 1
Concurrency 2
Arrays (abandoned)
2D Graphics
TSes: feature branches for beta release & then merge
TS bars start and end where work on detailed specification wording starts ("adopt initial working draft") and ends ("send for final ballot / publication")
Future starts/ends are shaded to indicate that dates and TS branches are approximate and subject to change

Online resources

- http://www.isocpp.org/
- http://en.cppreference.com/
- http://gcc.godbolt.org/
- http://coliru.stacked-crooked.com/

Objects

- The constructs in a C++ program create, destroy, refer to, access, and manipulate objects
- An object is a region of storage
  - has a storage duration/lifetime
  - has a type
  - can have a name

Objects

- The constructs in a C++ program create, destroy, refer to, access, and manipulate objects
- An object is a region of storage
  - has a storage duration/lifetime
  - has a type
  - can have a name

Objects

- The constructs in a C++ program create, destroy, refer to, access, and manipulate objects
- An object is a region of storage
  - has a storage duration/lifetime
  - has a type
  - can have a name

Objects

- The constructs in a C++ program create, destroy, refer to, access, and manipulate objects
- An object is a region of storage
  - has a storage duration/lifetime
  - has a type
  - can have a name

Objects

- The constructs in a C++ program create, destroy, refer to, access, and manipulate objects
- An object is a region of storage
  - has a storage duration/lifetime
  - has a type
  - can have a name

What is a type

- A type identifies a set of values and the operations that can be applied to those values
- C++ defines a few fundamental types and provides mechanisms to build compound and user-defined types on top of them
- A type is also associated with a machine representation for the values belonging to the type

What is a type

- A type identifies a set of values and the operations that can be applied to those values
- C++ defines a few fundamental types and provides mechanisms to build compound and user-defined types on top of them
- A type is also associated with a machine representation for the values belonging to the type

What is a type

- A type identifies a set of values and the operations that can be applied to those values
- C++ defines a few fundamental types and provides mechanisms to build compound and user-defined types on top of them
- A type is also associated with a machine representation for the values belonging to the type

Fundamental types

- arithmetic types
  - integral types
    - character types: `char`, `signed char`, `unsigned char`
    - signed integer types: `short int`, `int`, `long int`, `long long int`
    - unsigned integer types: `unsigned short int`, `unsigned int`, `unsigned long int`, `unsigned long long int`
    - `bool`
  - floating-point types: `float`, `double`, `long double`
- `std::nullptr_t`
- `void`

In general, size and machine representation are not defined by the standard.

Fundamental types

- arithmetic types
  - integral types
    - character types: `char`, `signed char`, `unsigned char`
    - signed integer types: `short int`, `int`, `long int`, `long long int`
    - unsigned integer types: `unsigned short int`, `unsigned int`, `unsigned long int`, `unsigned long long int`
    - `bool`
  - floating-point types: `float`, `double`, `long double`
- `std::nullptr_t`
- `void`

In general, size and machine representation are not defined by the standard.

int

Type representing a signed integer number

- set of values: subset of $\mathbb{Z}$
- operations: `+` `-` `*` `/` `%` `++` `--` `=` `==` `!=` `<` `>` ...
- representation: (usually) 2's complement
  - with N bits, values are in the range $[-2^{N-1}, 2^{N-1} - 1]$
  - integer overflow causes **undefined behavior**

```
// on my laptop
static_assert(sizeof(int) == 4);
```

int

Type representing a signed integer number

- set of values: subset of $\mathbb{Z}$
- operations: `+ - * / % ++ -- = == != < >` ...
- representation: (usually) 2's complement
  - with N bits, values are in the range $[-2^{N-1}, 2^{N-1} - 1]$
  - integer overflow causes **undefined behavior**

```
// on my laptop
static_assert(sizeof(int) == 4);
```

Literals

A literal is a constant value of a certain type included in the source code

- integer
- floating point
- character
- string
- boolean
- `nullptr`

A mechanism exists to introduce user-defined literals

- `123.5km 34j .5s`

Literals

A literal is a constant value of a certain type included in the source code

- integer
- floating point
- character
- string
- boolean
- `nullptr`

A mechanism exists to introduce user-defined literals

- `123.5km 34j .5s`

Literals

A literal is a constant value of a certain type included in the source code

- integer
- floating point
- character
- string
- boolean
- `nullptr`

A mechanism exists to introduce user-defined literals

- `123.5km 34j .5s`

Integer literals

decimal non-`0` decimal digit followed by zero or more digits

- `1 -98 123456789 -1'234'567'890`

octal `0` followed by octal digits

- `01 -077 07'654'321`

exadecimal `0x` or `0X` followed by hexadecimal digits

- `-0xdead 0xDEad123f 0XdeAD'123F`

binary `0b` or `0B` followed by binary digits

- `0b1101111010101101`
  `0B1101'1110'1010'1101`

- The default type for an integer literal is signed `int`
- Suffixes can be added to express unsignedness and size
  - `123u 123U`
  - `123l -123L 12311 -9223372036854775800LL`
  - `123ul 1231u`

Integer literals

decimal non-`0` decimal digit followed by zero or more digits

- `1 -98 123456789 -1'234'567'890`

octal `0` followed by octal digits

- `01 -077 07'654'321`

exadecimal `0x` or `0X` followed by hexadecimal digits

- `-0xdead 0xDEad123f 0XdeAD'123F`

binary `0b` or `0B` followed by binary digits

- `0b1101111010101101`
  `0B1101'1110'1010'1101`

- The default type for an integer literal is signed `int`
- Suffixes can be added to express unsignedness and size
  - `123u 123U`
  - `123l -123L 12311 -92233720368547758001L`
  - `123ul 1231u`

Integer literals

decimal non-`0` decimal digit followed by zero or more digits

- `1 -98 123456789 -1'234'567'890`

octal `0` followed by octal digits

- `01 -077 07'654'321`

exadecimal `0x` or `0X` followed by hexadecimal digits

- `-0xdead 0xDEad123f 0XdeAD'123F`

binary `0b` or `0B` followed by binary digits

- `0b1101111010101101`
- `0B1101'1110'1010'1101`

- The default type for an integer literal is signed `int`
- Suffixes can be added to express unsignedness and size
  - `123u 123U`
  - `123l -123L 12311 -9223372036854775800LL`
  - `123ul 1231u`

Integer literals

decimal non-`0` decimal digit followed by zero or more digits

- `1 -98 123456789 -1'234'567'890`

octal `0` followed by octal digits

- `01 -077 07'654'321`

exadecimal `0x` or `0X` followed by hexadecimal digits

- `-0xdead 0xDEad123f 0XdeAD'123F`

binary `0b` or `0B` followed by binary digits

- `0b1101111010101101`
  `0B1101'1110'1010'1101`

- The default type for an integer literal is signed `int`
- Suffixes can be added to express unsignedness and size
  - `123u 123U`
  - `123l -123L 12311 -9223372036854775800LL`
  - `123ul 1231u`

Integer literals

decimal non-`0` decimal digit followed by zero or more digits

- `1 -98 123456789 -1'234'567'890`

octal `0` followed by octal digits

- `01 -077 07'654'321`

exadecimal `0x` or `0X` followed by hexadecimal digits

- `-0xdead 0xDEad123f 0XdeAD'123F`

binary `0b` or `0B` followed by binary digits

- `0b1101111010101101`
  `0B1101'1110'1010'1101`

- The default type for an integer literal is signed `int`
- Suffixes can be added to express unsignedness and size
  - `123u 123U`
  - `123l -123L 12311 -9223372036854775800LL`
  - `123ul 1231u`

Integer literals

decimal non-`0` decimal digit followed by zero or more digits

- `1 -98 123456789 -1'234'567'890`

octal `0` followed by octal digits

- `01 -077 07'654'321`

exadecimal `0x` or `0X` followed by hexadecimal digits

- `-0xdead 0xDEad123f 0XdeAD'123F`

binary `0b` or `0B` followed by binary digits

- `0b1101111010101101`
  `0B1101'1110'1010'1101`

- The default type for an integer literal is signed `int`
- Suffixes can be added to express unsignedness and size
  - `123u 123U`
  - `123l -123L 123ll -9223372036854775800LL`
  - `123ul 123lu`

Floating-point literals

significand optional-exponent optional-suffix

significand a sequence of decimal digits with or without a decimal separator

- `12.34 -12. 12 -.34`

optional-exponent an `e` or `E` followed by a signed decimal integer number

- `e0 E3 e-3`

optional-suffix specifies the size (the default is `double`)

- `float` or `long double`
- `f F l L`

Putting it all together

- `12.34e3 -.34e-3 12.34e3f -1234e-2L`

Boolean literals

- `true` and `false`
- The type of a boolean literal is `bool`

Character literals

- Let's assume a character literal is an ASCII character between single quotes
  - `'a' 'B' '7' '#' '/'`
- Some ASCII characters need to be expressed as `\`-escaped sequences
  - `'\'' '\\' '\n' '\t' '\0'`
- The type of a character literal is `char`

Character literals

- Let's assume a character literal is an ASCII character between single quotes
  - `'a' 'B' '7' '#' '/'`
- Some ASCII characters need to be expressed as `\`-escaped sequences
  - `'\'' '\\' '\n' '\t' '\0'`
- The type of a character literal is `char`

Character literals

- Let's assume a character literal is an ASCII character between single quotes
  - `'a' 'B' '7' '#' '/'`
- Some ASCII characters need to be expressed as `\`-escaped sequences
  - `'\'' '\\' '\n' '\t' '\0'`
- The type of a character literal is `char`

String literals

- A string literal is either a sequence of escaped or non-escaped characters between double quotes . . .
  - `"hello" "hello\tworld" "hello \"world\""`
  - a `'\0'` is implicitly appended at the end, so it cannot be part of the sequence
- . . . or any character sequence between `R"`*`delimiter`*`(` and `)`*`delimiter`*`"`
  - `R"(hello "world")" R":(hello "(world)"):"`
  - raw-string literal
  - useful for the manipulation of e.g. regular expressions or XML text
- The type of a string literal is `char const[]`

String literals

- A string literal is either a sequence of escaped or non-escaped characters between double quotes . . .
  - `"hello" "hello\tworld" "hello \"world\""`
  - a `'\0'` is implicitly appended at the end, so it cannot be part of the sequence
- . . . or any character sequence between `R"`*`delimiter`*`(` and `)`*`delimiter`*`"`
  - `R"(hello "world")" R":(hello "(world)"):"`
  - raw-string literal
  - useful for the manipulation of e.g. regular expressions or XML text
- The type of a string literal is `char const[]`

String literals

- A string literal is either a sequence of escaped or non-escaped characters between double quotes . . .
  - `"hello" "hello\tworld" "hello \"world\""`
  - a `'\0'` is implicitly appended at the end, so it cannot be part of the sequence
- . . . or any character sequence between `R"`*`delimiter`*`(` and `)`*`delimiter`*`"`
  - `R"(hello "world")" R":(hello "(world)"):"`
  - raw-string literal
  - useful for the manipulation of e.g. regular expressions or XML text
- The type of a string literal is `char const[]`

Variables

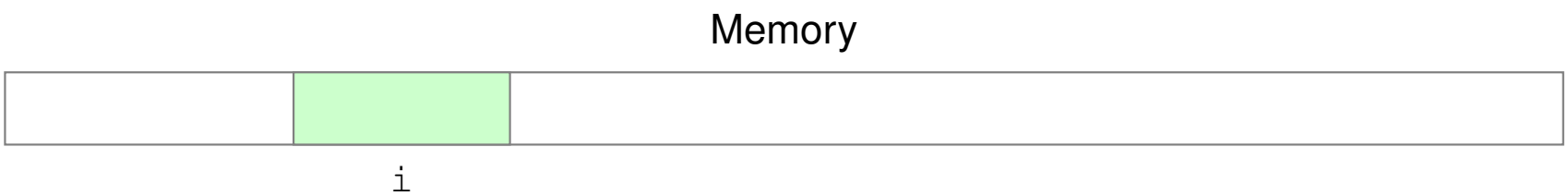
- A variable is a name for an *object*
- A name is an *identifier*: a sequence of letters (including _) and digits, starting with a letter

```
int i;              // declaration; the value is undefined
i = 4321;           // assignment
int j = 1234;       // declaration and initialization
i = j;              // assignment of j's value to i
assert(i == j); // i and j have the same value
```

Variables

- A variable is a name for an *object*
- A name is an *identifier*: a sequence of letters (including _) and digits, starting with a letter

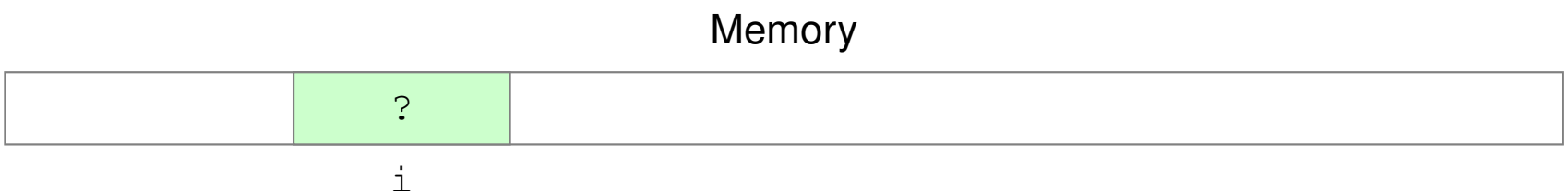
```
int i;           // declaration; the value is undefined
i = 4321;        // assignment
int j = 1234;    // declaration and initialization
i = j;           // assignment of j's value to i
assert(i == j); // i and j have the same value
```



Variables

- A variable is a name for an *object*
- A name is an *identifier*: a sequence of letters (including _) and digits, starting with a letter

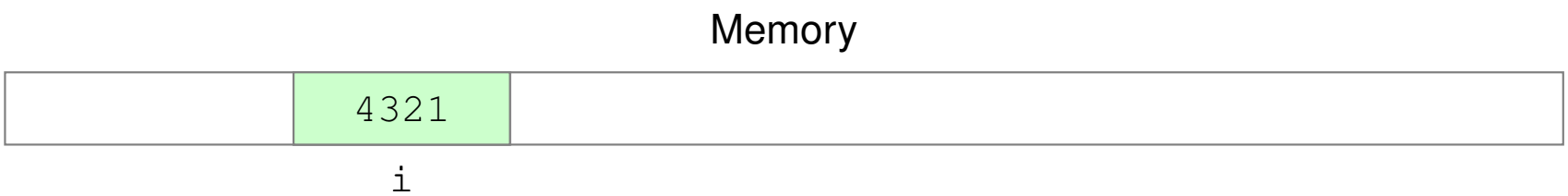
```
int i;           // declaration; the value is undefined
i = 4321;        // assignment
int j = 1234;    // declaration and initialization
i = j;           // assignment of j's value to i
assert(i == j); // i and j have the same value
```



Variables

- A variable is a name for an *object*
- A name is an *identifier*: a sequence of letters (including _) and digits, starting with a letter

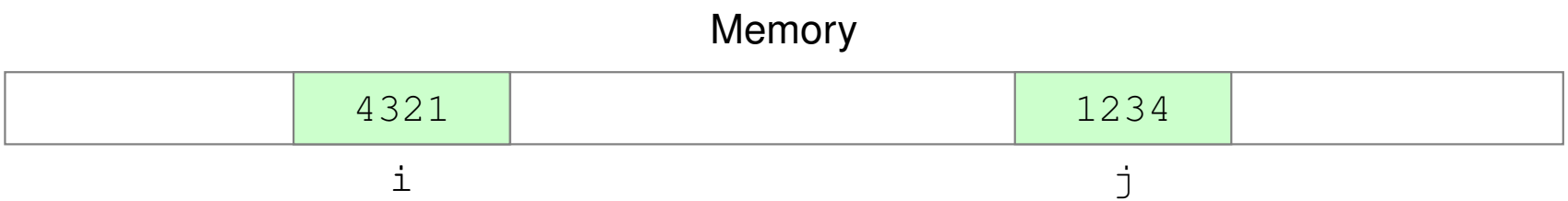
```
int i;            // declaration; the value is undefined
i = 4321;         // assignment
int j = 1234;     // declaration and initialization
i = j;            // assignment of j's value to i
assert(i == j); // i and j have the same value
```



Variables

- A variable is a name for an *object*
- A name is an *identifier*: a sequence of letters (including _) and digits, starting with a letter

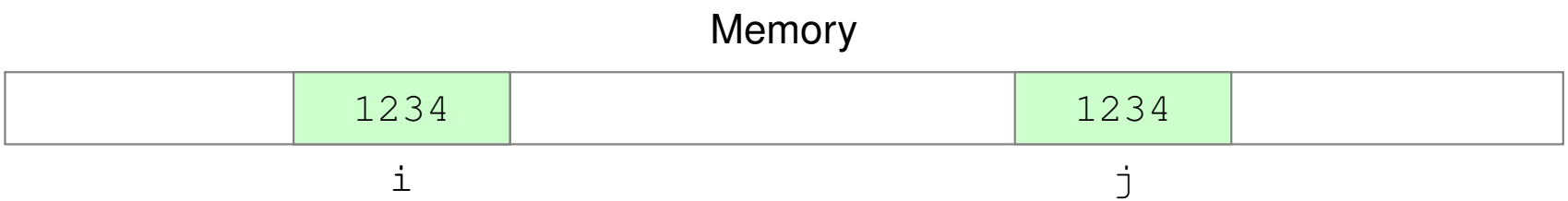
```
int i;           // declaration; the value is undefined
i = 4321;        // assignment
int j = 1234;    // declaration and initialization
i = j;           // assignment of j's value to i
assert(i == j); // i and j have the same value
```



Variables

- A variable is a name for an *object*
- A name is an *identifier*: a sequence of letters (including _) and digits, starting with a letter

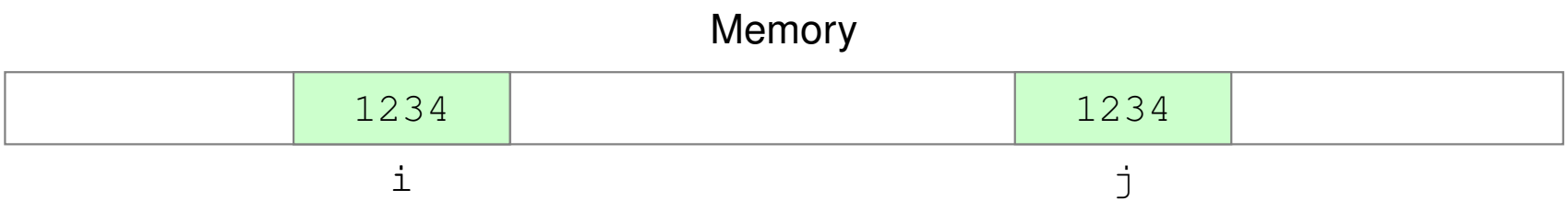
```
int i;           // declaration; the value is undefined
i = 4321;        // assignment
int j = 1234;    // declaration and initialization
i = j;           // assignment of j's value to i
assert(i == j); // i and j have the same value
```



Variables

- A variable is a name for an *object*
- A name is an *identifier*: a sequence of letters (including _) and digits, starting with a letter

```
int i;           // declaration; the value is undefined
i = 4321;        // assignment
int j = 1234;    // declaration and initialization
i = j;           // assignment of j's value to i
assert(i == j); // i and j have the same value
```



Keywords

The following identifiers are reserved

```
alignas     const          for          private             throw
alignof     constexpr      friend       protected           true
and         const_cast     goto         public              try
and_eq      continue       if           register            typedef
asm         decltype       inline       reinterpret_cast    typeid
auto        default        int          return              typename
bitand      delete         long         short               union
bitor       do             mutable      signed              unsigned
bool        double         namespace    sizeof              using
break       dynamic_cast   new          static              virtual
case        else           noexcept     static_assert       void
catch       enum           not          static_cast         volatile
char        explicit       not_eq       struct              xor
char16_t    export         nullptr      switch              xor_eq
char32_t    extern         operator     template            wchar_t
class       false          or           this                while
compl       float          or_eq        thread_local
```

Expressions

- An expression is a sequence of operators and their operands that specifies a computation
- Literals and variables are typical operands, but there are others

```
1 + 2
i = 1 + 2
i == j
sqrt(x) > 1.42
std::cout << "hello, " << str << '\n'
```

Some expressions have *side-effects*

Expressions

- An expression is a sequence of operators and their operands that specifies a computation
- Literals and variables are typical operands, but there are others

```
1 + 2
i = 1 + 2
i == j
sqrt(x) > 1.42
std::cout << "hello, " << str << '\n'
```

Some expressions have *side-effects*

Expressions

- An expression is a sequence of operators and their operands that specifies a computation
- Literals and variables are typical operands, but there are others

```
1 + 2
i = 1 + 2
i == j
sqrt(x) > 1.42
std::cout << "hello, " << str << '\n'
```

Some expressions have *side-effects*

Expressions

- An expression is a sequence of operators and their operands that specifies a computation
- Literals and variables are typical operands, but there are others

```
1 + 2
i = 1 + 2
i == j
sqrt(x) > 1.42
std::cout << "hello, " << str << '\n'
```

Some expressions have *side-effects*

Expressions

- An expression is a sequence of operators and their operands that specifies a computation
- Literals and variables are typical operands, but there are others

```
1 + 2
i = 1 + 2
i == j
sqrt(x) > 1.42
std::cout << "hello, " << str << '\n'
```

Some expressions have *side-effects*

Expressions

- An expression is a sequence of operators and their operands that specifies a computation
- Literals and variables are typical operands, but there are others

```
1 + 2
i = 1 + 2
i == j
sqrt(x) > 1.42
std::cout << "hello, " << str << '\n'
```

Some expressions have *side-effects*

Expressions

- An expression is a sequence of operators and their operands that specifies a computation
- Literals and variables are typical operands, but there are others

```
1 + 2
i = 1 + 2
i == j
sqrt(x) > 1.42
std::cout << "hello, " << str << '\n'
```

Some expressions have *side-effects*

Operators

Arithmetic

```
+a
-a
a + b
a - b
a * b
a / b
a % b
~a
a & b
a | b
a ^ b
a << b
a >> b
```

Logical

```
!a
a && b
a || b
```

Comparison

```
a == b
a != b
a < b
a > b
a <= b
a >= b
```

Increment

```
++a
--a
a++
a--
```

Assignments

```
a = b
a += b
a -= b
a *= b
a /= b
a %= b
a &= b
a |= b
a ^= b
a <<= b
a >>= b
```

Access

```
a[b]
*a
&a
a->b
a.b
```

Other

```
a(...)
a, b
? :
```

Plus: casts, allocation and deallocation, static introspection, . . .

- Rules exist for associativity, commutativity and precedence
- Many operators can be overloaded for user-defined types
  - e.g. + to sum complex numbers

Operators

Arithmetic

```
+a
-a
a + b
a - b
a * b
a / b
a % b
~a
a & b
a | b
a ^ b
a << b
a >> b
```

Logical

```
!a
a && b
a || b
```

Comparison

```
a == b
a != b
a < b
a > b
a <= b
a >= b
```

Increment

```
++a
--a
a++
a--
```

Assignments

```
a = b
a += b
a -= b
a *= b
a /= b
a %= b
a &= b
a |= b
a ^= b
a <<= b
a >>= b
```

Access

```
a[b]
*a
&a
a->b
a.b
```

Other

```
a(...)
a, b
? :
```

Plus: casts, allocation and deallocation, static introspection, . . .

- Rules exist for associativity, commutativity and precedence
- Many operators can be overloaded for user-defined types
  - e.g. + to sum complex numbers

Operators

Arithmetic

```
+a
-a
a + b
a - b
a * b
a / b
a % b
~a
a & b
a | b
a ^ b
a << b
a >> b
```

Logical

```
!a
a && b
a || b
```

Comparison

```
a == b
a != b
a < b
a > b
a <= b
a >= b
```

Increment

```
++a
--a
a++
a--
```

Assignments

```
a = b
a += b
a -= b
a *= b
a /= b
a %= b
a &= b
a |= b
a ^= b
a <<= b
a >>= b
```

Access

```
a[b]
*a
&a
a->b
a.b
```

Other

```
a(...)
a, b
? :
```

Plus: casts, allocation and deallocation, static introspection, . . .

- Rules exist for associativity, commutativity and precedence
- Many operators can be overloaded for user-defined types
  - e.g. + to sum complex numbers

Type conversions

- A value of type `T1` may be converted implicitly to a value of type `T2` in order to match the expected type in a certain situation

```
1 + 2.3
```

  - between numbers and bool
  - between signed and unsigned numbers
  - between numbers of different size
  - between integral and floating-point numbers
  - ...

- Conversions sometimes are surprising
- Conversions can be explicit using `static_cast`

```
1 + static_cast<int>(2.3)
```

- Mechanims exist to define implicit and explicit conversions involving user-defined types

Type conversions

- A value of type `T1` may be converted implicitly to a value of type `T2` in order to match the expected type in a certain situation

```
1 + 2.3
```

  - between numbers and bool
  - between signed and unsigned numbers
  - between numbers of different size
  - between integral and floating-point numbers
  - ...

- Conversions sometimes are surprising
- Conversions can be explicit using `static_cast`

```
1 + static_cast<int>(2.3)
```

- Mechanims exist to define implicit and explicit conversions involving user-defined types

Type conversions

- A value of type `T1` may be converted implicitly to a value of type `T2` in order to match the expected type in a certain situation

```
1 + 2.3
```

  - between numbers and bool
  - between signed and unsigned numbers
  - between numbers of different size
  - between integral and floating-point numbers
  - ...

- Conversions sometimes are surprising
- Conversions can be explicit using `static_cast`

```
1 + static_cast<int>(2.3)
```

- Mechanims exist to define implicit and explicit conversions involving user-defined types

Type conversions

- A value of type `T1` may be converted implicitly to a value of type `T2` in order to match the expected type in a certain situation

```
1 + 2.3
```

  - between numbers and bool
  - between signed and unsigned numbers
  - between numbers of different size
  - between integral and floating-point numbers
  - ...

- Conversions sometimes are surprising
- Conversions can be explicit using `static_cast`

```
1 + static_cast<int>(2.3)
```

- Mechanims exist to define implicit and explicit conversions involving user-defined types

Type conversions

- A value of type `T1` may be converted implicitly to a value of type `T2` in order to match the expected type in a certain situation

```
1 + 2.3
```

  - between numbers and bool
  - between signed and unsigned numbers
  - between numbers of different size
  - between integral and floating-point numbers
  - ...

- Conversions sometimes are surprising
- Conversions can be explicit using `static_cast`

```
1 + static_cast<int>(2.3)
```

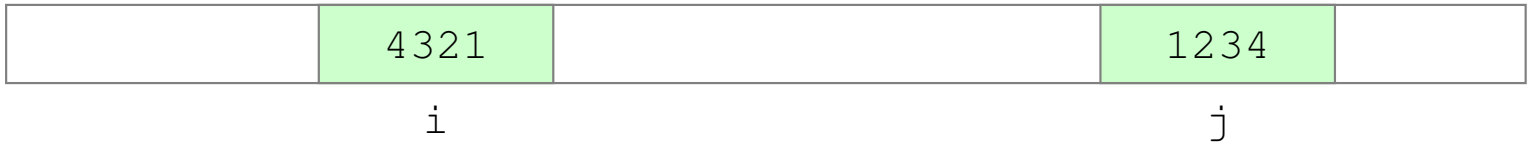
- Mechanims exist to define implicit and explicit conversions involving user-defined types

Pointers

Where in memory does a given object reside?

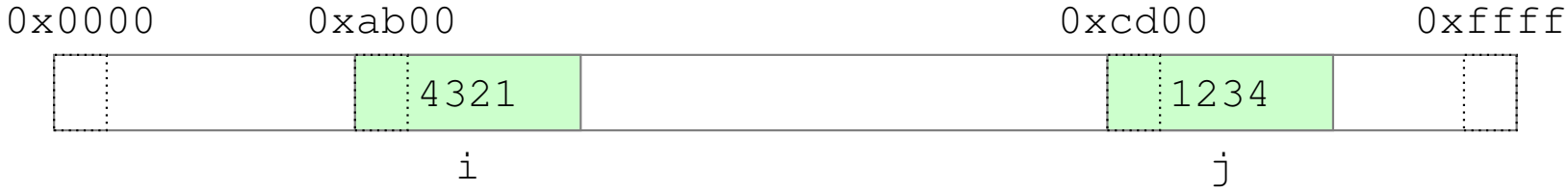
Pointers

Where in memory does a given object reside?



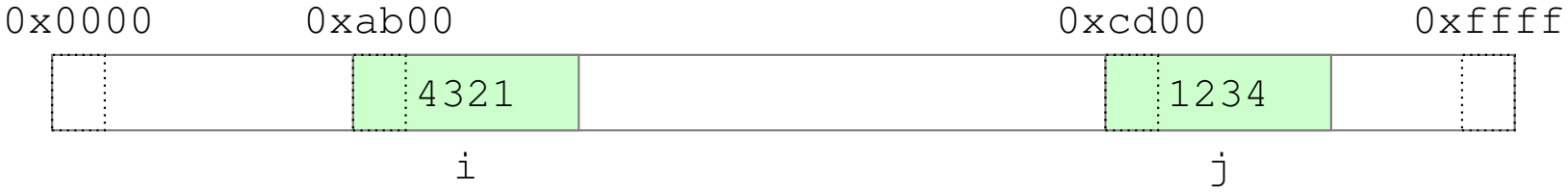
Pointers

Where in memory does a given object reside?



Pointers

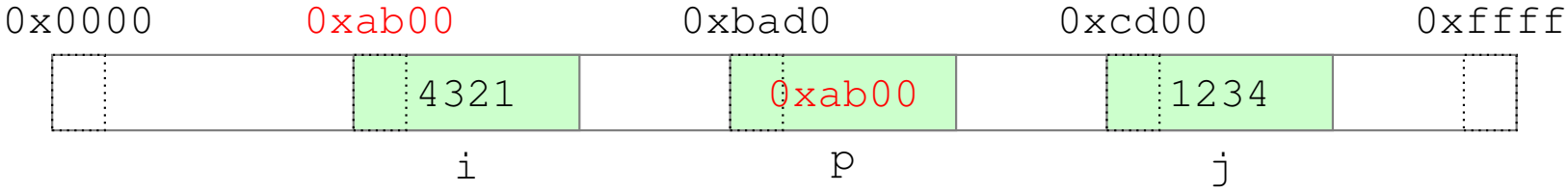
Where in memory does a given object reside?




```
assert(&i == 0xab00); // address-of operator
assert(&j == 0xcd00);
```

Pointers

Where in memory does a given object reside?

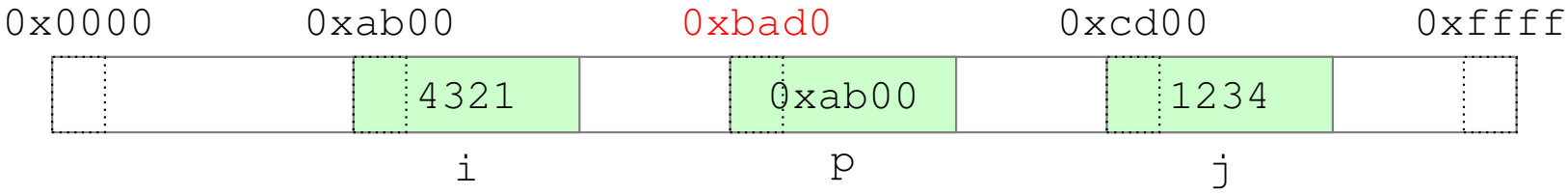



```
assert(&i == 0xab00); // address-of operator
assert(&j == 0xcd00);
int* p = &i; // pointer declarator
```

Note that `int*` is a type by itself

Pointers

Where in memory does a given object reside?

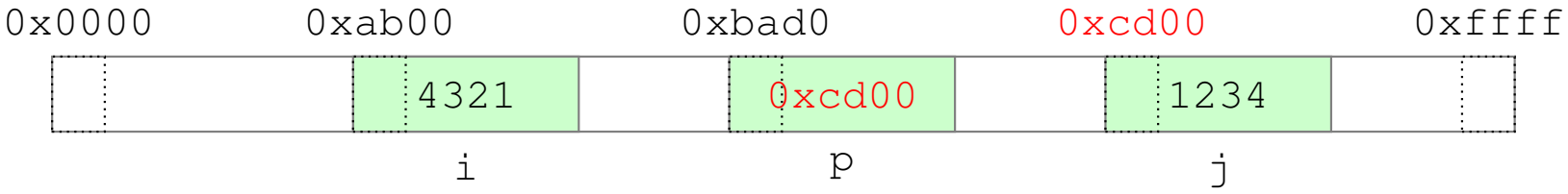



```
assert(&i == 0xab00); // address-of operator
assert(&j == 0xcd00);
int* p = &i; // pointer declarator
assert(&p == 0xbad0);
int** pp = &p; // &p is of type int**
```

Note that `int*` is a type by itself and so is `int**`

Pointers

Where in memory does a given object reside?

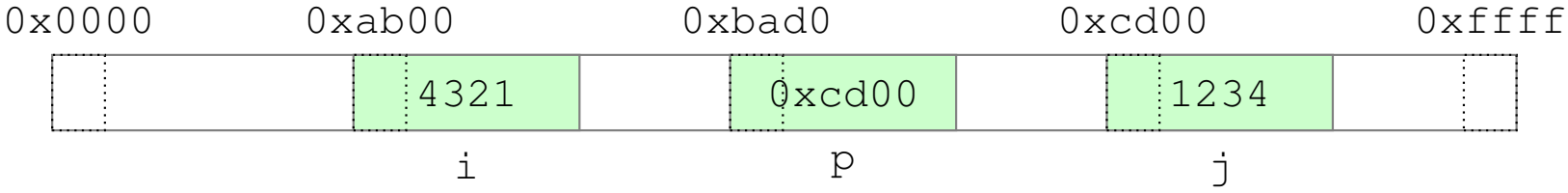



```
assert(&i == 0xab00); // address-of operator
assert(&j == 0xcd00);
int* p = &i; // pointer declarator
assert(&p == 0xbad0);
int** pp = &p; // &p is of type int**
p = &j;
```

Note that `int*` is a type by itself and so is `int**`

Pointers

Where in memory does a given object reside?

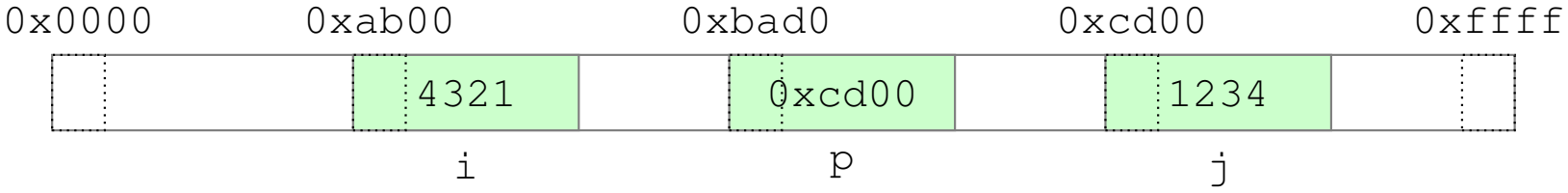



```
assert(&i == 0xab00); // address-of operator
assert(&j == 0xcd00);
int* p = &i; // pointer declarator
assert(&p == 0xbad0);
int** pp = &p; // &p is of type int**
p = &j;
int* q = p; // p and q point to the same object
```

Note that `int*` is a type by itself and so is `int**`

Pointers

Where in memory does a given object reside?



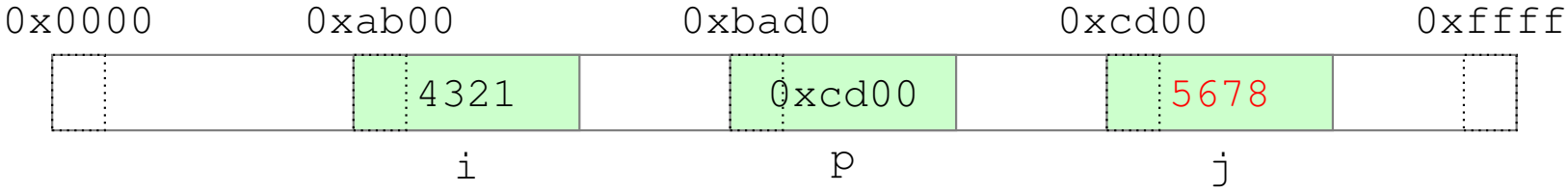

```
assert(&i == 0xab00); // address-of operator
assert(&j == 0xcd00);
int* p = &i; // pointer declarator
assert(&p == 0xbad0);
int** pp = &p; // &p is of type int**
p = &j;
int* q = p; // p and q point to the same object
```

```
assert(*p == j); // dereference operator
int k = *p;
```

Note that `int*` is a type by itself and so is `int**`

Pointers

Where in memory does a given object reside?



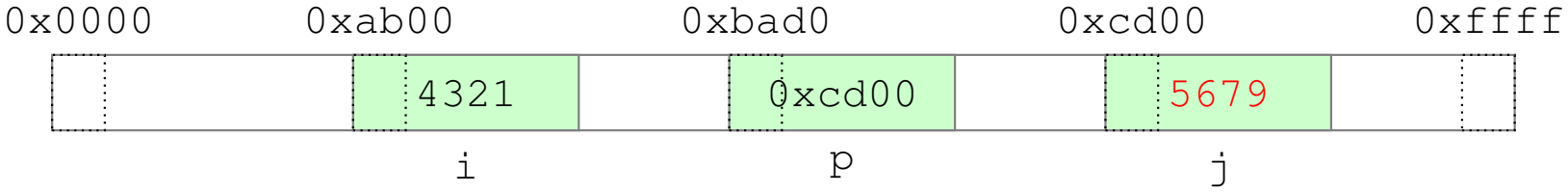

```
assert(&i == 0xab00); // address-of operator
assert(&j == 0xcd00);
int* p = &i; // pointer declarator
assert(&p == 0xbad0);
int** pp = &p; // &p is of type int**
p = &j;
int* q = p; // p and q point to the same object
```

```
assert(*p == j); // dereference operator
int k = *p;
*p = 5678;
```

Note that `int*` is a type by itself and so is `int**`

Pointers

Where in memory does a given object reside?



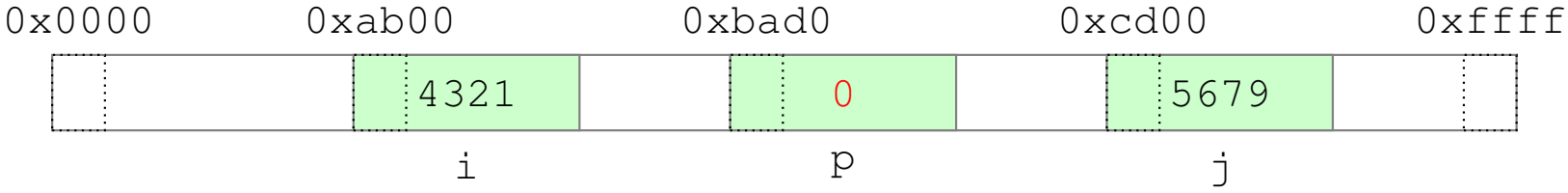

```
assert(&i == 0xab00); // address-of operator
assert(&j == 0xcd00);
int* p = &i; // pointer declarator
assert(&p == 0xbad0);
int** pp = &p; // &p is of type int**
p = &j;
int* q = p; // p and q point to the same object
```

```
assert(*p == j); // dereference operator
int k = *p;
*p = 5678;
++(*p);
```

Note that `int*` is a type by itself and so is `int**`

Pointers

Where in memory does a given object reside?



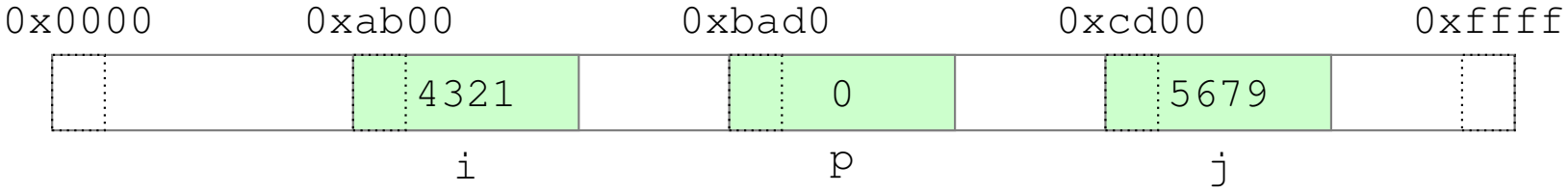

```
assert(&i == 0xab00); // address-of operator
assert(&j == 0xcd00);
int* p = &i; // pointer declarator
assert(&p == 0xbad0);
int** pp = &p; // &p is of type int**
p = &j;
int* q = p; // p and q point to the same object
```

```
assert(*p == j); // dereference operator
int k = *p;
*p = 5678;
++(*p);
p = nullptr;
```

Note that `int*` is a type by itself and so is `int**`

Pointers

Where in memory does a given object reside?




```
assert(&i == 0xab00); // address-of operator
assert(&j == 0xcd00);
int* p = &i; // pointer declarator
assert(&p == 0xbad0);
int** pp = &p; // &p is of type int**
p = &j;
int* q = p; // p and q point to the same object
```

```
assert(*p == j); // dereference operator
int k = *p;
*p = 5678;
++(*p);
p = nullptr;
*p; // undefined behavior
```

Note that `int*` is a type by itself and so is `int**`

(lvalue) References

A reference is another name (*alias*) for an existing object

- must be initialized to refer to a valid object when declared
- there is no such thing as a null reference

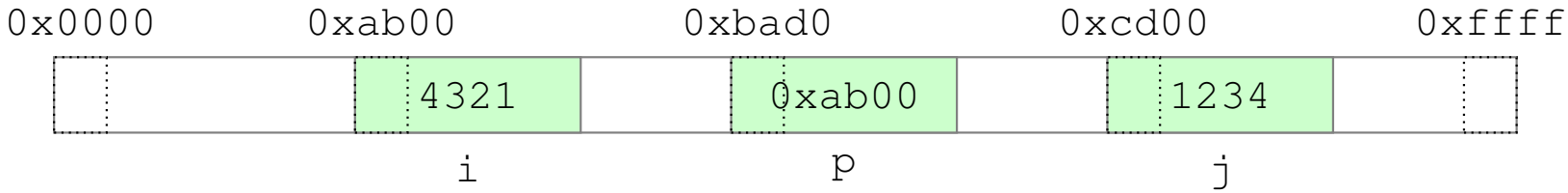
```
int& ri = i; // reference declarator
assert(ri == i);
assert(&ri == &i);
ri = 6789;
ri = j;
ri = *p; // *p indeed provides a reference to the object
```

`int&` is a type by itself

(lvalue) References

A reference is another name (*alias*) for an existing object

- must be initialized to refer to a valid object when declared
- there is no such thing as a null reference



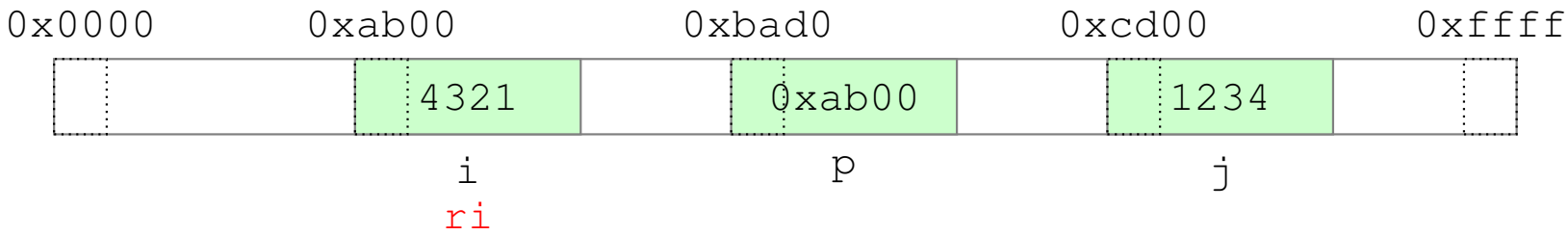

```
int& ri = i; // reference declarator
assert(ri == i);
assert(&ri == &i);
ri = 6789;
ri = j;
ri = *p; // *p indeed provides a reference to the object
```

`int&` is a type by itself

(lvalue) References

A reference is another name (*alias*) for an existing object

- must be initialized to refer to a valid object when declared
- there is no such thing as a null reference



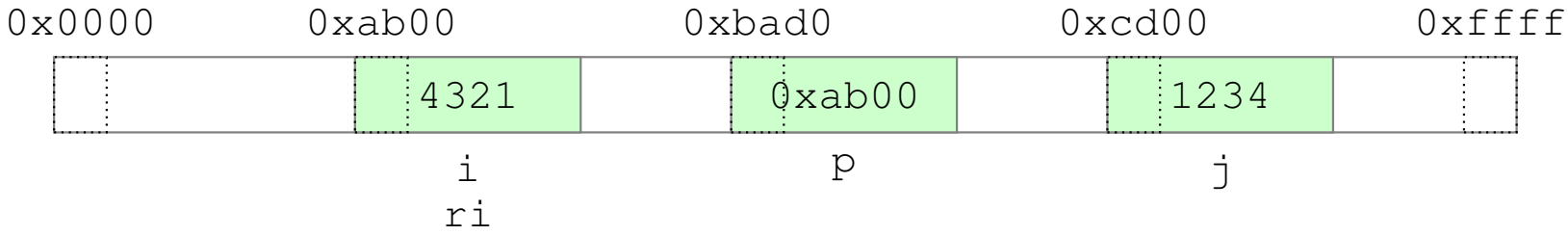

```
int& ri = i; // reference declarator
assert(ri == i);
assert(&ri == &i);
ri = 6789;
ri = j;
ri = *p; // *p indeed provides a reference to the object
```

`int&` is a type by itself

(lvalue) References

A reference is another name (*alias*) for an existing object

- must be initialized to refer to a valid object when declared
- there is no such thing as a null reference



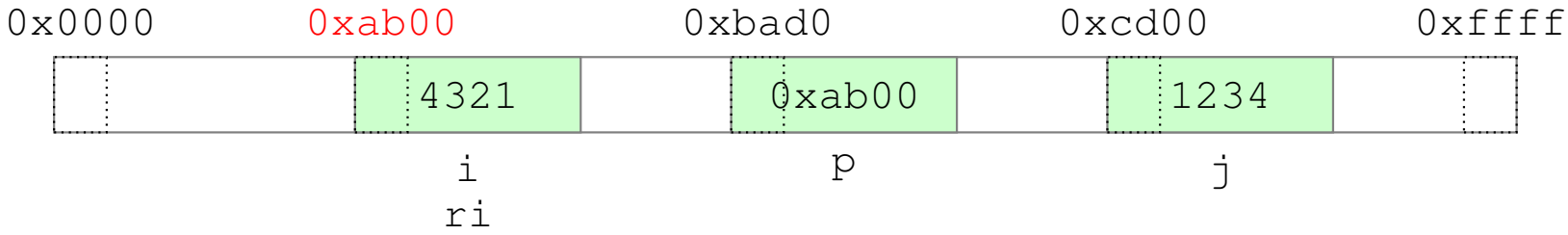

```
int& ri = i; // reference declarator
assert(ri == i);
assert(&ri == &i);
ri = 6789;
ri = j;
ri = *p; // *p indeed provides a reference to the object
```

`int&` is a type by itself

(lvalue) References

A reference is another name (*alias*) for an existing object

- must be initialized to refer to a valid object when declared
- there is no such thing as a null reference



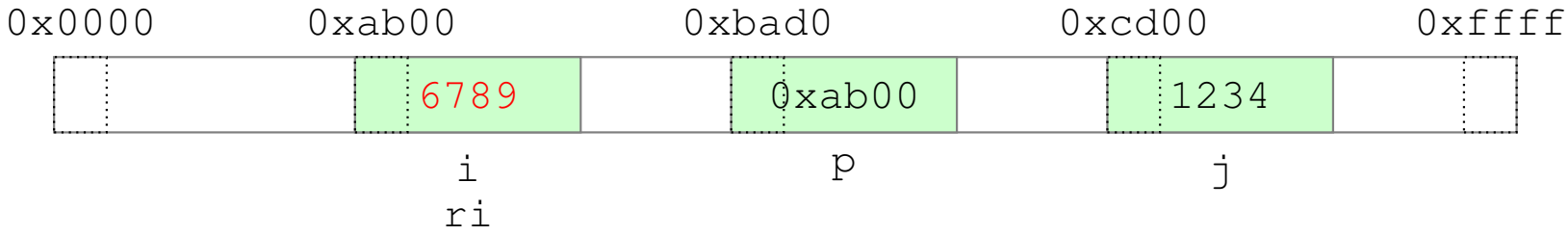

```
int& ri = i; // reference declarator
assert(ri == i);
assert(&ri == &i);
ri = 6789;
ri = j;
ri = *p; // *p indeed provides a reference to the object
```

`int&` is a type by itself

(lvalue) References

A reference is another name (*alias*) for an existing object

- must be initialized to refer to a valid object when declared
- there is no such thing as a null reference



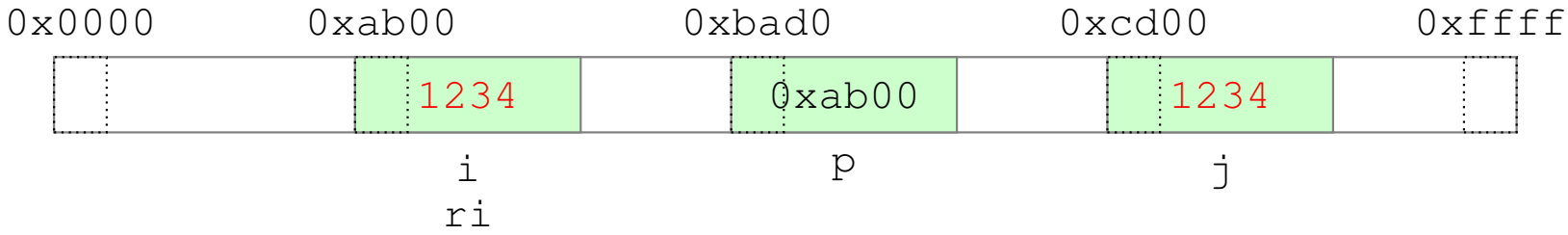

```
int& ri = i; // reference declarator
assert(ri == i);
assert(&ri == &i);
ri = 6789;
ri = j;
ri = *p; // *p indeed provides a reference to the object
```

`int&` is a type by itself

(lvalue) References

A reference is another name (*alias*) for an existing object

- must be initialized to refer to a valid object when declared
- there is no such thing as a null reference



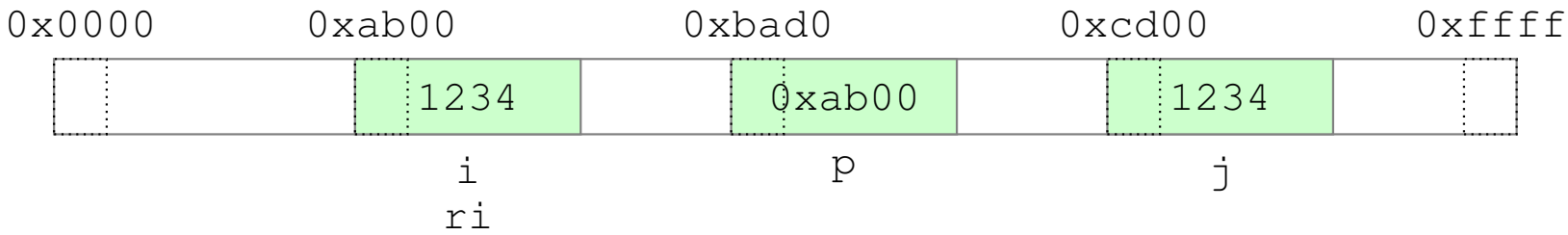

```
int& ri = i; // reference declarator
assert(ri == i);
assert(&ri == &i);
ri = 6789;
ri = j;
ri = *p; // *p indeed provides a reference to the object
```

`int&` is a type by itself

(lvalue) References

A reference is another name (*alias*) for an existing object

- must be initialized to refer to a valid object when declared
- there is no such thing as a null reference



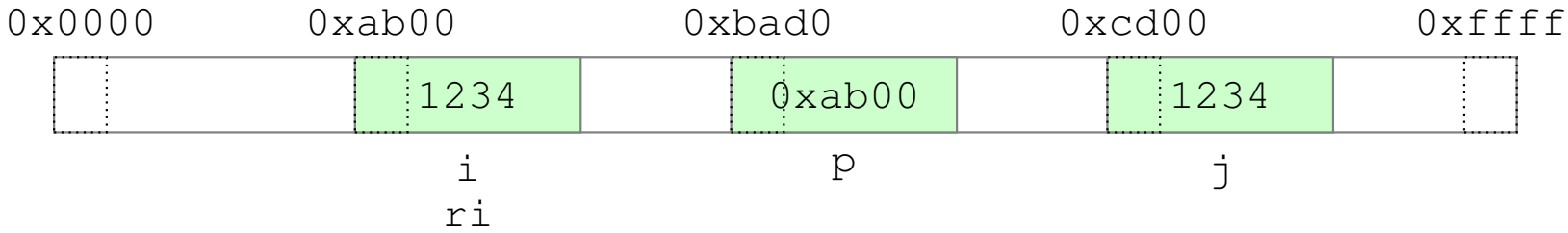

```
int& ri = i; // reference declarator
assert(ri == i);
assert(&ri == &i);
ri = 6789;
ri = j;
ri = *p; // *p indeed provides a reference to the object
```

`int&` is a type by itself

(lvalue) References

A reference is another name (*alias*) for an existing object

- must be initialized to refer to a valid object when declared
- there is no such thing as a null reference




```
int& ri = i; // reference declarator
assert(ri == i);
assert(&ri == &i);
ri = 6789;
ri = j;
ri = *p; // *p indeed provides a reference to the object
```

`int&` is a type by itself

Comments

Comments can be included in code in two ways

- text between `/*` and `*/`, possibly on multiple lines
- text after `//` until the end of the line

```
int /* this is a comment */ k;
int f; // this is a comment
/* start of a multi-line comment
k = 5;
int& r = k; // this is a nested comment
*/
```

Comments are treated by the compiler as spaces

const-correctness

- Data qualified as `const` is logically immutable
- Data that is meant to be immutable should be `const`

```
int const k;          // error
int const cvar = 5; // or `const int`
cvar * 2;
cvar = 3;             // error
```

```
int& r = cvar;             // error
int const& cvarr = cvar; // or `const int&`
int var;
int const& varr = var;   // read-only view of var
int& const cr = var;      // error
```

```
int* p = &cvar;             // error
int const* cvarp = &cvar; // or `const int*`
int const* varp = &var;
```

```
// cp is const but var can be modified thru cp
int* const cp = &var;
// ccp is const and var cannot be modified thru ccp
int const* const ccp = &var;
```

const-correctness

- Data qualified as `const` is logically immutable
- Data that is meant to be immutable should be `const`

```
int const k;          // error
int const cvar = 5; // or `const int`
cvar * 2;
cvar = 3;             // error
```

```
int& r = cvar;             // error
int const& cvarr = cvar; // or `const int&`
int var;
int const& varr = var;   // read-only view of var
int& const cr = var;      // error
```

```
int* p = &cvar;             // error
int const* cvarp = &cvar; // or `const int*`
int const* varp = &var;
```

```
// cp is const but var can be modified thru cp
int* const cp = &var;
// ccp is const and var cannot be modified thru ccp
int const* const ccp = &var;
```

const-correctness

- Data qualified as `const` is logically immutable
- Data that is meant to be immutable should be `const`

```
int const k;         // error
int const cvar = 5; // or `const int`
cvar * 2;
cvar = 3;            // error
```

```
int& r = cvar;            // error
int const& cvarr = cvar; // or `const int&`
int var;
int const& varr = var;   // read-only view of var
int& const cr = var;     // error
```

```
int* p = &cvar;            // error
int const* cvarp = &cvar; // or `const int*`
int const* varp = &var;
```

```
// cp is const but var can be modified thru cp
int* const cp = &var;
// ccp is const and var cannot be modified thru ccp
int const* const ccp = &var;
```

const-correctness

- Data qualified as `const` is logically immutable
- Data that is meant to be immutable should be `const`

```
int const k;         // error
int const cvar = 5; // or `const int`
cvar * 2;
cvar = 3;            // error
```

```
int& r = cvar;            // error
int const& cvarr = cvar; // or `const int&`
int var;
int const& varr = var;   // read-only view of var
int& const cr = var;     // error
```

```
int* p = &cvar;            // error
int const* cvarp = &cvar; // or `const int*`
int const* varp = &var;
```

```
// cp is const but var can be modified thru cp
int* const cp = &var;
// ccp is const and var cannot be modified thru ccp
int const* const ccp = &var;
```

const-correctness

- Data qualified as const is logically immutable
- Data that is meant to be immutable should be const

```
int const k;          // error
int const cvar = 5; // or `const int`
cvar * 2;
cvar = 3;             // error
```

```
int& r = cvar;             // error
int const& cvarr = cvar; // or `const int&`
int var;
int const& varr = var;   // read-only view of var
int& const cr = var;      // error
```

```
int* p = &cvar;             // error
int const* cvarp = &cvar; // or `const int*`
int const* varp = &var;
```

```
// cp is const but var can be modified thru cp
int* const cp = &var;
// ccp is const and var cannot be modified thru ccp
int const* const ccp = &var;
```

auto

Let the compiler deduce the type of a variable from the initializer

```
auto i = 0;                 // int
auto u = 0U;                // unsigned int
auto p = &i;                // int*
auto const d = 1.;          // double const
std::vector<std::string> v;
auto it = std::begin(v); // std::vector<std::string>::iterator
auto e;                     // error
```

`auto` never deduces a reference

```
auto& r = i;          // int&
auto rr = r;          // int
auto const& cr = i; // int const&
auto& crr = cr;       // int const&
```

Statement

Statements are units of code that are executed in sequence

- expression statements
- compound statements
- selection statements
- iteration statements
- jump statements
- declaration statements
- try blocks

Expression statement

An expression followed by a semicolon (`;`)

The value of the expression (if any) is discarded

The expression can have side effects

```
b + 2;
a = b + 2;
; // empty statement
```

Expression statement

An expression followed by a semicolon (`;`)
The value of the expression (if any) is discarded
The expression can have side effects

```
b + 2;
a = b + 2;
; // empty statement
```

Expression statement

An expression followed by a semicolon (`;`)
The value of the expression (if any) is discarded
The expression can have side effects

```
b + 2;
a = b + 2;
; // empty statement
```

Expression statement

An expression followed by a semicolon (`;`)
The value of the expression (if any) is discarded
The expression can have side effects

```
b + 2;
a = b + 2;
; // empty statement
```

Compound statement (or block)

A sequence of zero or more statements enclosed between braces ({ })

```
{
    found = true;
    ++i;
}
```

```
{ found = true; ++i; } // all on one line
```

Declaration statement

A declaration statement introduces one or more new identifiers into a block

A declaration of a variable in a block makes the variable of automatic storage duration

- the variable is initialized each time the declaration is executed
- the variable is destroyed each time the execution reaches the end of the block

Declaration statement

A declaration statement introduces one or more new identifiers into a block
A declaration of a variable in a block makes the variable of automatic storage duration

- the variable is initialized each time the declaration is executed
- the variable is destroyed each time the execution reaches the end of the block

Scope

The scope of a name appearing in a program is the, possibly discontiguous, portion of source code where that name is valid

- block scope
- class scope
- namespace scope
- ...

Block scope

The scope of a name declared in a block starts at the declaration and ends at the }

```
{
  ...
  int n = 0; // scope of n begins
  ++n;
} // scope of n ends
++n; // error
```

The scope of a variable should be as small as possible

- Don't declare all variables at the beginning of the block
- Declare a variable only when there is a value to initialize it

Block scope

The scope of a name declared in a block starts at the declaration and ends at the }

```
{
  ...
  int n = 0; // scope of n begins
  ++n;
} // scope of n ends
++n; // error
```

The scope of a variable should be as small as possible

- Don't declare all variables at the beginning of the block
- Declare a variable only when there is a value to initialize it

Block scope

The scope of a name declared in a block starts at the declaration and ends at the }

```
{
  ...
  int n = 0; // scope of n begins
  ++n;
} // scope of n ends
++n; // error
```

The scope of a variable should be as small as possible

- Don't declare all variables at the beginning of the block
- Declare a variable only when there is a value to initialize it

Functions

- A function associates a block of statements with
  - a name
  - a list of zero or more parameters
- A function may return a value

```
void print(double d)
{ ... }

double rand()
{ ... }

double pow(double x, double y)
{ ... }

Configuration read_config(boost::filesystem::path const& p)
{ ... }
```

Functions

- A function associates a block of statements with
  - a name
  - a list of zero or more parameters
- A function may return a value

```
void print(double d)
{ ... }

double rand()
{ ... }

double pow(double x, double y)
{ ... }

Configuration read_config(boost::filesystem::path const& p)
{ ... }
```

Functions

- A function associates a block of statements with
  - a name
  - a list of zero or more parameters
- A function may return a value

```
void print(double d)
{ ... }

double rand()
{ ... }

double pow(double x, double y)
{ ... }

Configuration read_config(boost::filesystem::path const& p)
{ ... }
```

Function call

```
void print(double d) { ... }

double rand() { ... }

void test_rand()
{
    auto a = rand(); // call `rand` with no arguments
    print(a);        // call `print` with argument `a`
    print(rand());   // all in one expression
    ...
}
```

Function call

```
void print(double d) { ... }

double rand() { ... }

void test_rand()
{
    auto a = rand(); // call ‘rand‘ with no arguments
    print(a);        // call ‘print‘ with argument ‘a‘
    print(rand());   // all in one expression
    ...
}
```

Function call

```
void print(double d) { ... }

double rand() { ... }

void test_rand()
{
    auto a = rand(); // call `rand` with no arguments
    print(a);        // call `print` with argument `a`
    print(rand());   // all in one expression
    ...
}
```

Function call

```
void print(double d) { ... }

double rand() { ... }

void test_rand()
{
    auto a = rand(); // call `rand` with no arguments
    print(a);        // call `print` with argument `a`
    print(rand());   // all in one expression
    ...
}
```

Function arguments

- During a call, function arguments are bound to function parameters position-wise
  - names are irrelevant
- Arguments can be passed
  - by value: a copy of the argument object is available inside the function body
  - by reference: an alias for the argument object is available inside the function body

Function arguments

- During a call, function arguments are bound to function parameters position-wise
  - names are irrelevant
- Arguments can be passed
  - by value: a copy of the argument object is available inside the function body
  - by reference: an alias for the argument object is available inside the function body

Function arguments

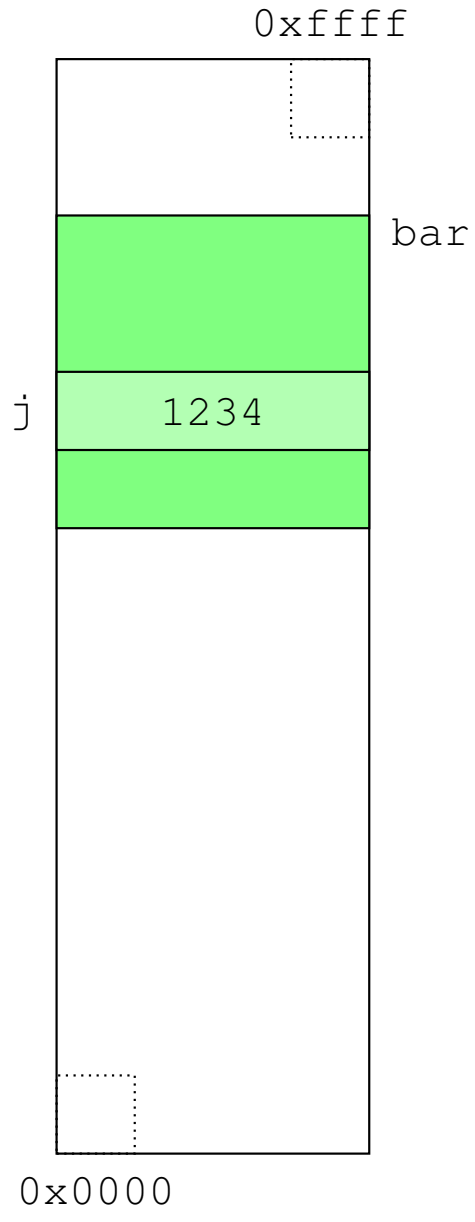
- During a call, function arguments are bound to function parameters position-wise
  - names are irrelevant
- Arguments can be passed
  - by value: a copy of the argument object is available inside the function body
  - by reference: an alias for the argument object is available inside the function body

Function arguments

- During a call, function arguments are bound to function parameters position-wise
  - names are irrelevant
- Arguments can be passed
  - by value: a copy of the argument object is available inside the function body
  - by reference: an alias for the argument object is available inside the function body

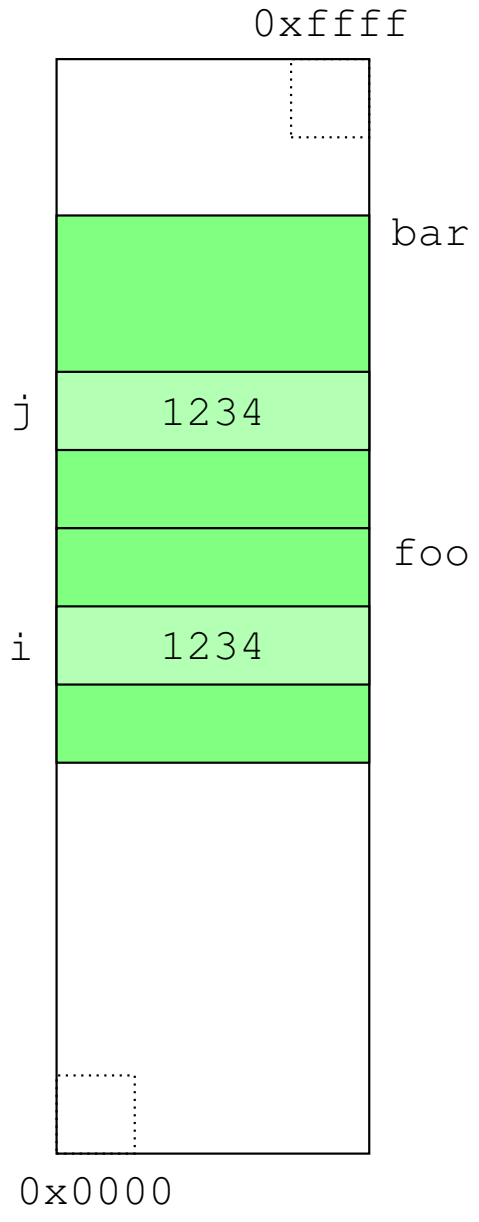
Pass by-value

```
void foo(int i)
{
  ++i;
}

void bar()
{
  int j = 1234;
  foo(j);
  assert(j == 1234);
}
```



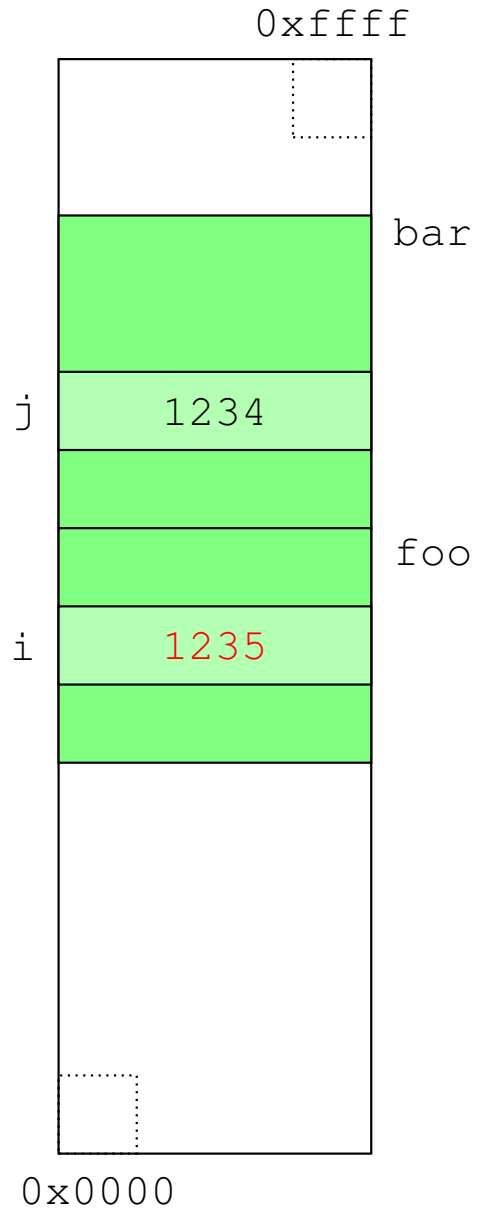
Pass by-value

```
void foo(int i)
{
  ++i;
}

void bar()
{
  int j = 1234;
  foo(j);
  assert(j == 1234);
}
```



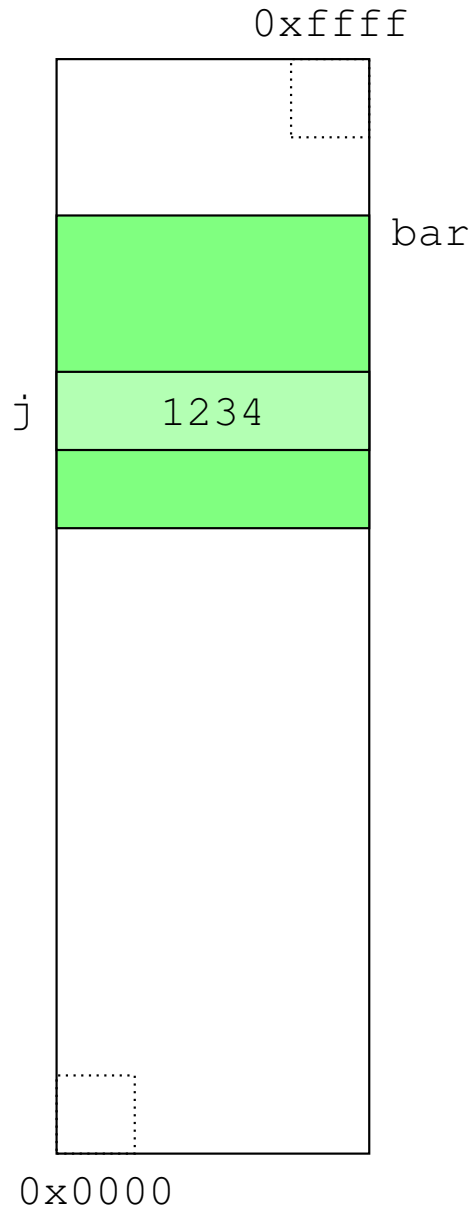
Pass by-value

```
void foo(int i)
{
  ++i;
}

void bar()
{
  int j = 1234;
  foo(j);
  assert(j == 1234);
}
```



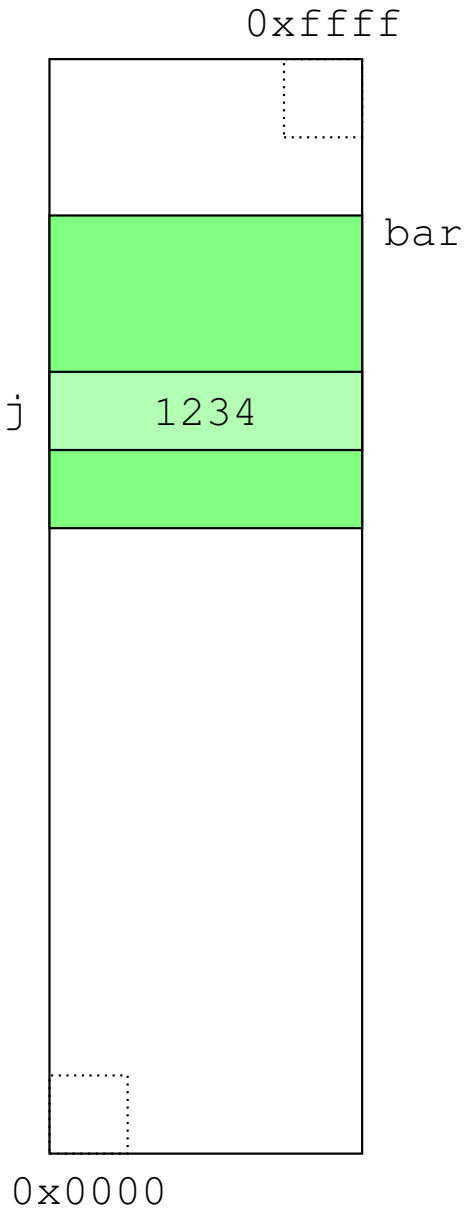
Pass by-value

```
void foo(int i)
{
  ++i;
}

void bar()
{
  int j = 1234;
  foo(j);
  assert(j == 1234);
}
```



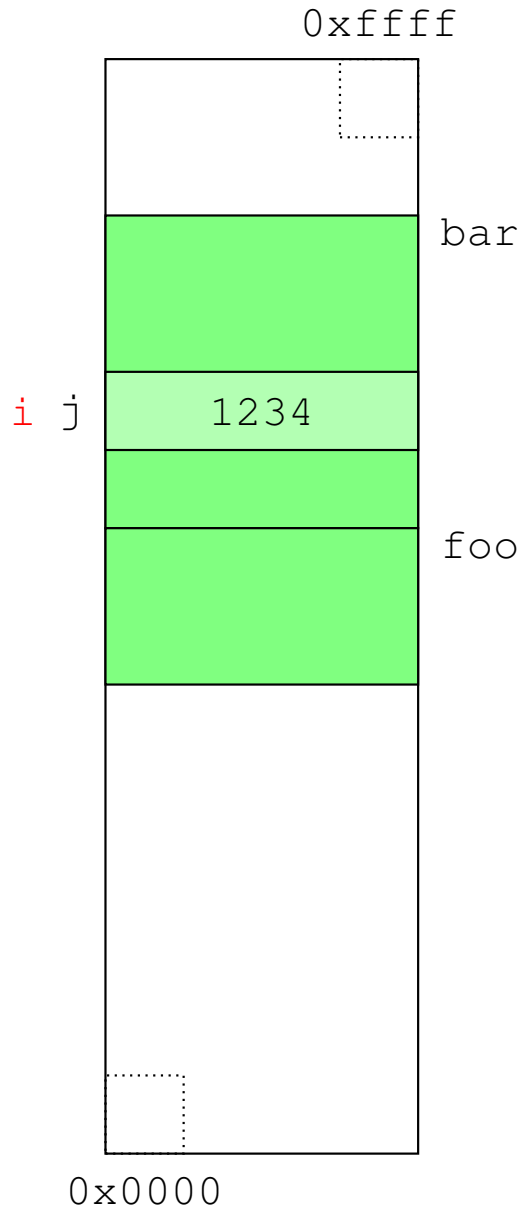
Pass by-reference

```
void foo(int& i)
{
  ++i;
}

void bar()
{
  int j = 1234;
  foo(j);
  assert(j == 1235);
}
```



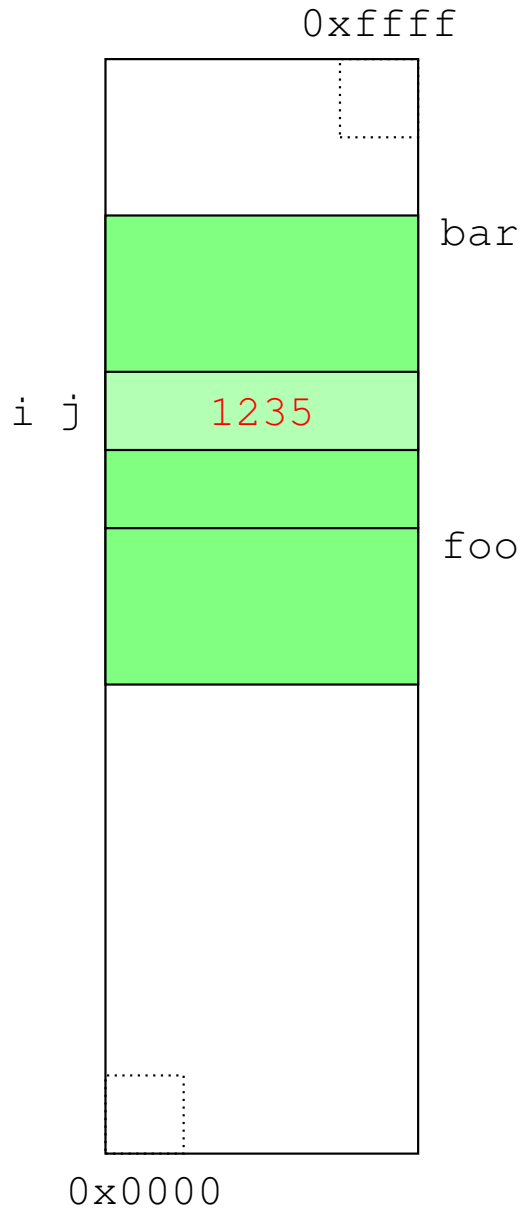
Pass by-reference

```
void foo(int& i)
{
  ++i;
}

void bar()
{
  int j = 1234;
  foo(j);
  assert(j == 1235);
}
```



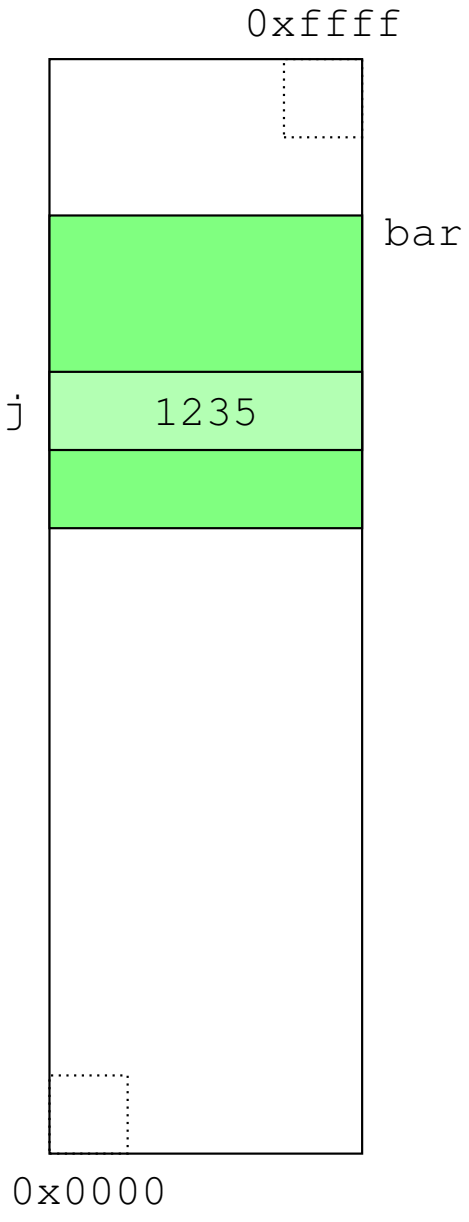
Pass by-reference

```
void foo(int& i)
{
  ++i;
}

void bar()
{
  int j = 1234;
  foo(j);
  assert(j == 1235);
}
```



Pass by-reference

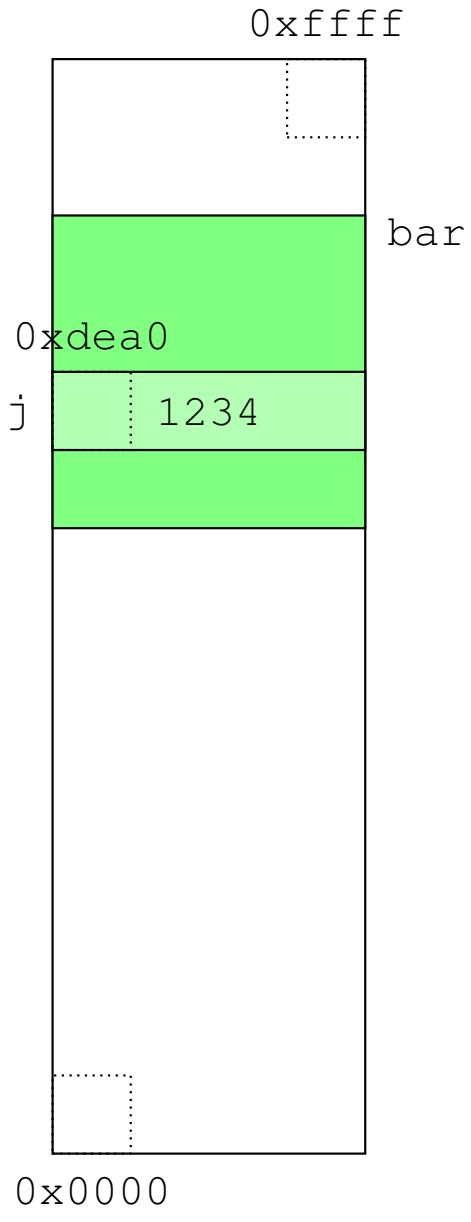
```
void foo(int& i)
{
  ++i;
}

void bar()
{
  int j = 1234;
  foo(j);
  assert(j == 1235);
}
```



Pass "by-pointer"

Actually a form of pass by-value

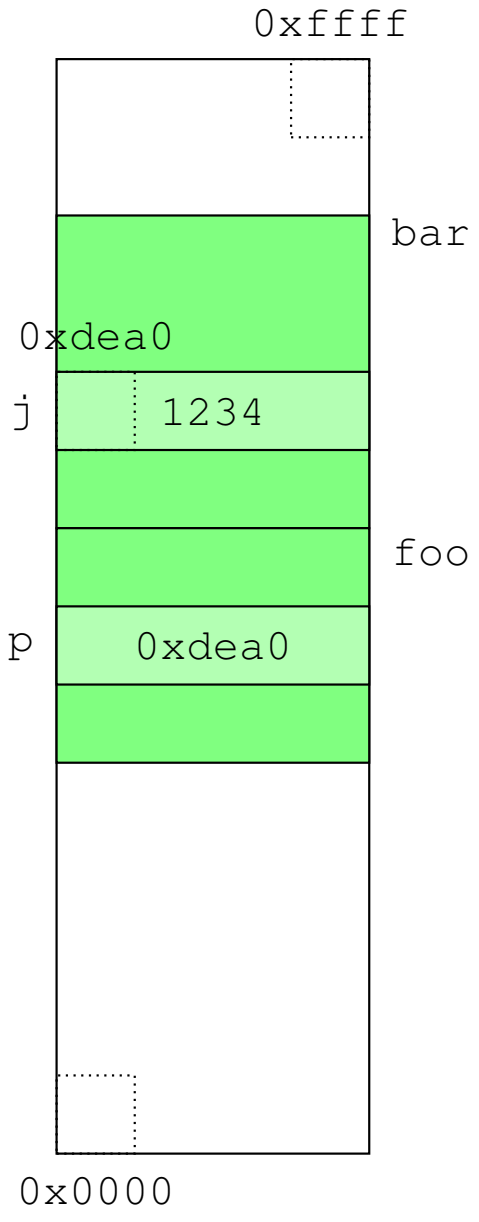
```
void foo(int* p)
{
  ++(*p);
}

void bar()
{
  int j = 1234;
  foo(&j);
  assert(j == 1235);
}
```



Pass "by-pointer"

Actually a form of pass by-value

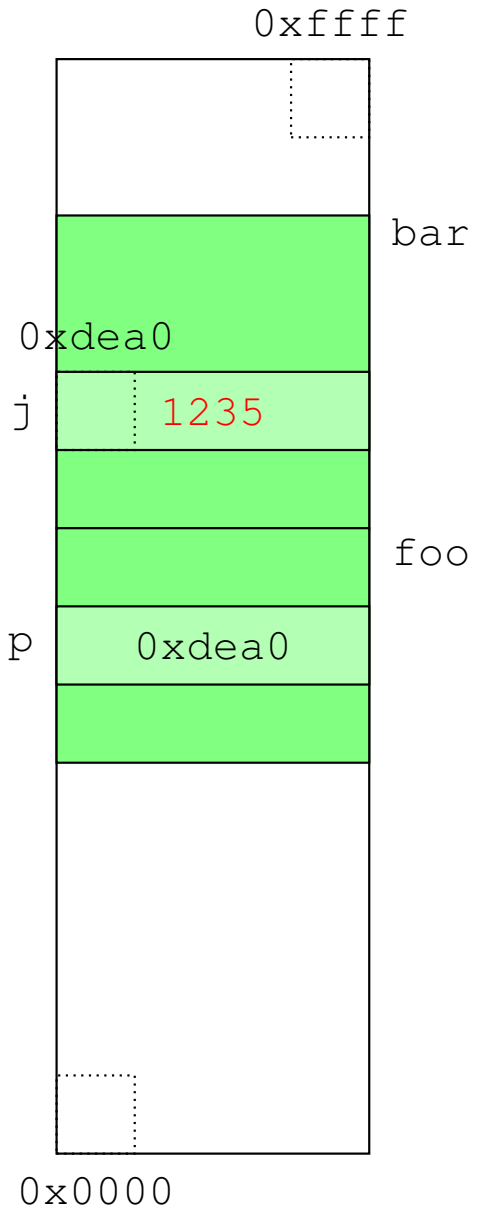
```
void foo(int* p)
{
  ++(*p);
}

void bar()
{
  int j = 1234;
  foo(&j);
  assert(j == 1235);
}
```



Pass "by-pointer"

Actually a form of pass by-value

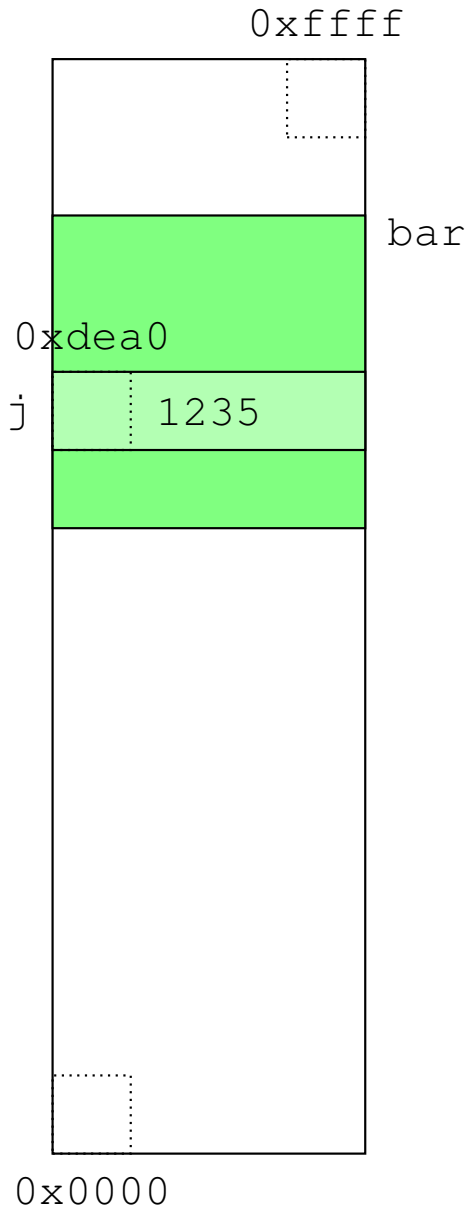
```
void foo(int* p)
{
  ++(*p);
}

void bar()
{
  int j = 1234;
  foo(&j);
  assert(j == 1235);
}
```



Pass "by-pointer"

Actually a form of pass by-value

```
void foo(int* p)
{
  ++(*p);
}

void bar()
{
  int j = 1234;
  foo(&j);
  assert(j == 1235);
}
```



Return from a function

- A function returns control to the caller when either
  - the execution reaches the function body closing brace `}`
  - there is and explicit `return` statement
- The `return` statement is mandatory when the function declaration contains a return type

```
void print(double) {
  ...
  return; // not necessary
}

double rand() {
  auto result = 0.;
  ...
  return 0.12311; // multiple return statements are possible
  ...
  return result;
}
```

Return from a function

- A function returns control to the caller when either
  - the execution reaches the function body closing brace `}`
  - there is and explicit `return` statement
- The `return` statement is mandatory when the function declaration contains a return type

```
void print(double) {
  ...
  return; // not necessary
}

double rand() {
  auto result = 0.;
  ...
  return 0.12311; // multiple return statements are possible
  ...
  return result;
}
```

Return from a function

- A function returns control to the caller when either
  - the execution reaches the function body closing brace `}`
  - there is and explicit `return` statement
- The `return` statement is mandatory when the function declaration contains a return type

```
void print(double) {
  ...
  return; // not necessary
}

double rand() {
  auto result = 0.;
  ...
  return 0.12311; // multiple return statements are possible
  ...
  return result;
}
```

Return from a function

- A function returns control to the caller when either
  - the execution reaches the function body closing brace `}`
  - there is and explicit `return` statement
- The `return` statement is mandatory when the function declaration contains a return type

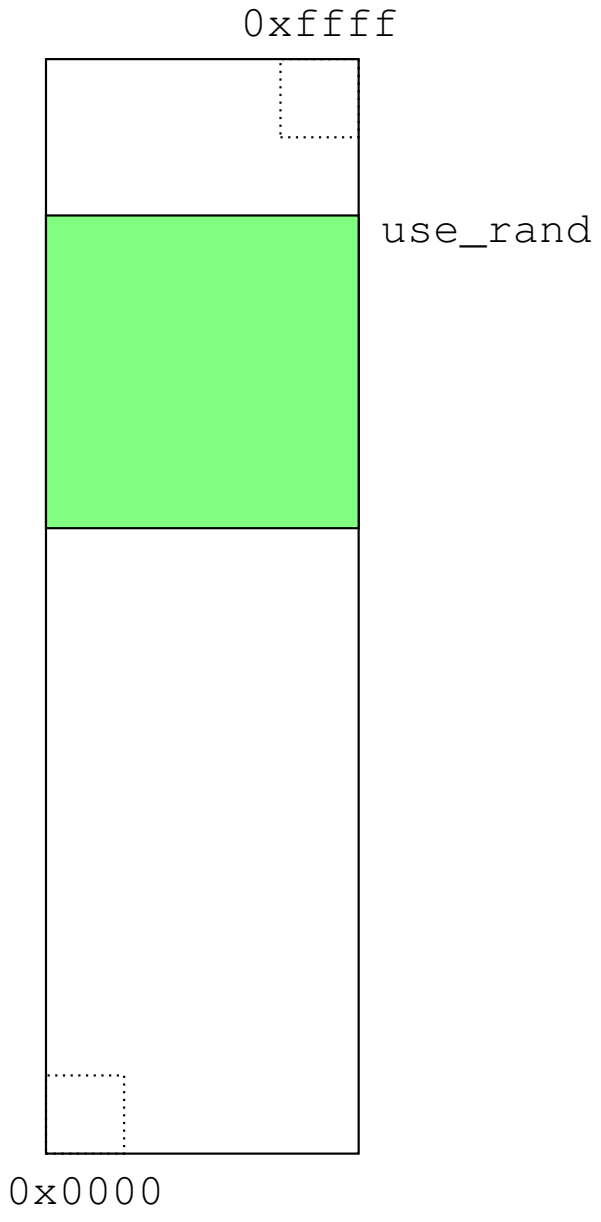
```
void print(double) {
  ...
  return; // not necessary
}

double rand() {
  auto result = 0.;
  ...
  return 0.12311; // multiple return statements are possible
  ...
  return result;
}
```

Return a value

```
double rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double r = rand();
  ...
}
```

The return value is copied (C++98) or moved (C++11 onwards) into the destination

- the copy/move can be optimized away

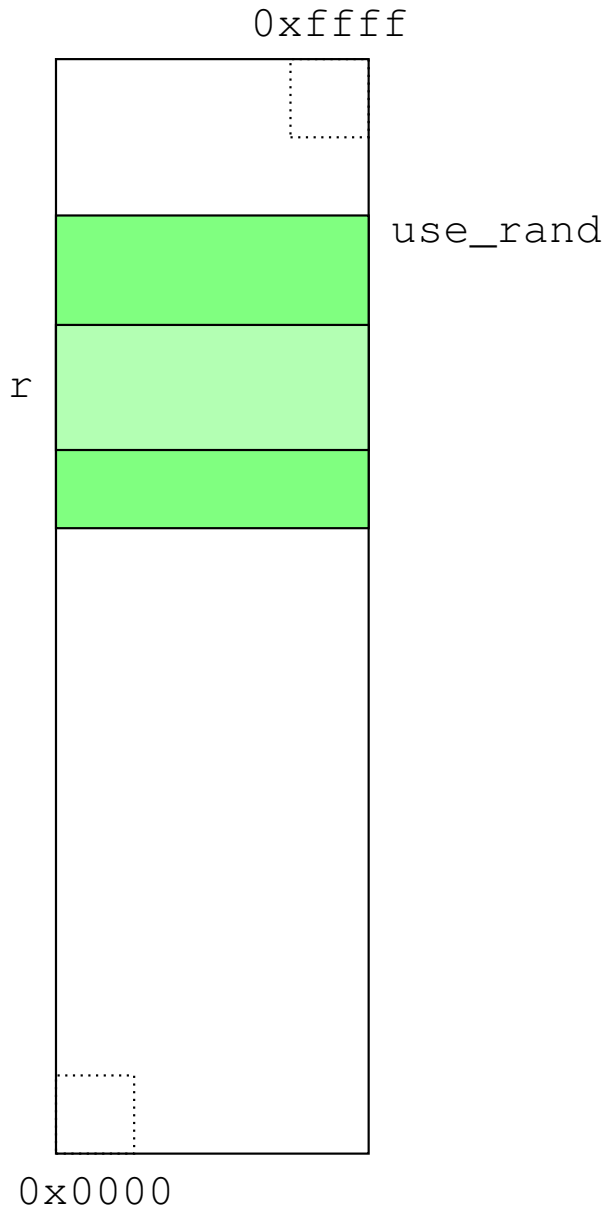


Return a value

```
double rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double r = rand();
  ...
}
```

The return value is copied (C++98) or moved (C++11 onwards) into the destination

- the copy/move can be optimized away

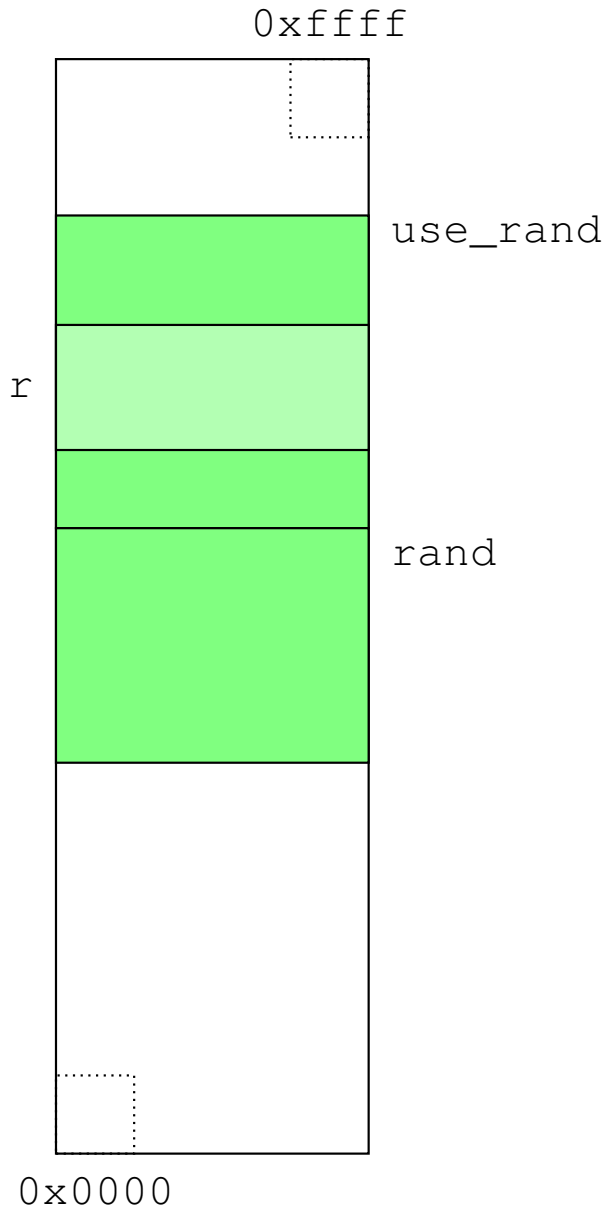


Return a value

```
double rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double r = rand();
  ...
}
```

The return value is copied (C++98) or moved (C++11 onwards) into the destination

- the copy/move can be optimized away

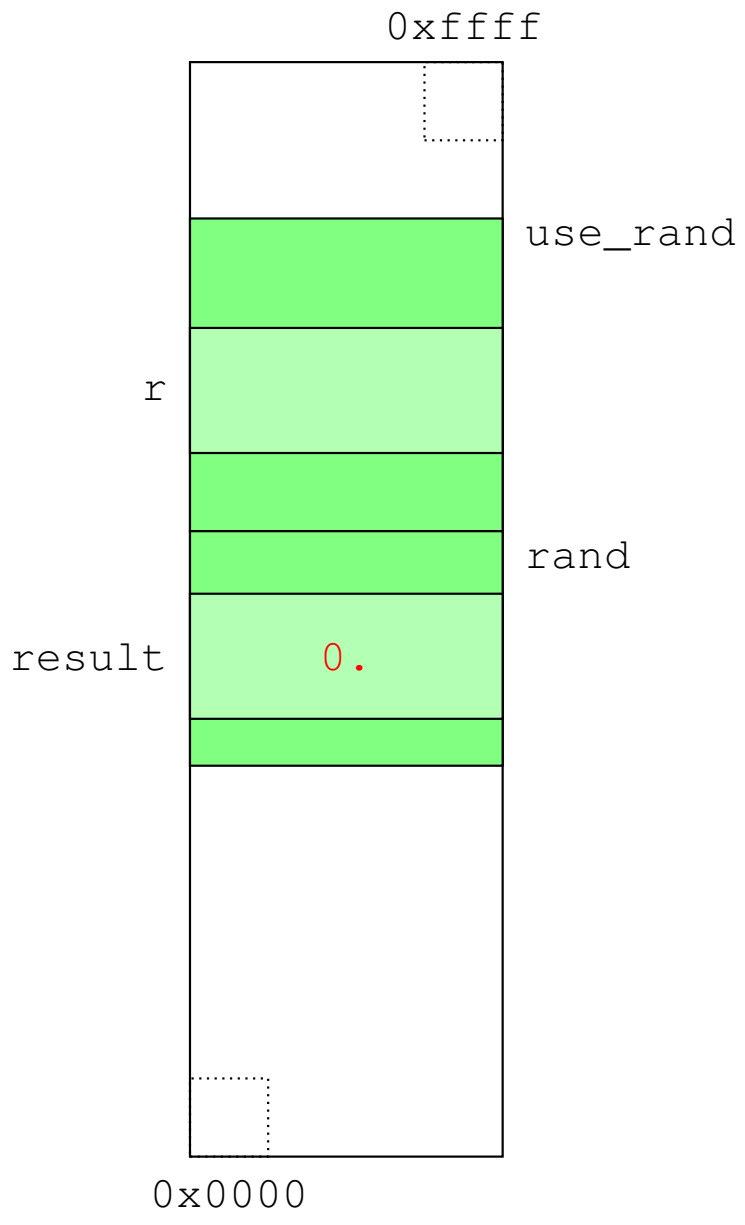


Return a value

```
double rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double r = rand();
  ...
}
```

The return value is copied (C++98) or moved (C++11 onwards) into the destination

- the copy/move can be optimized away

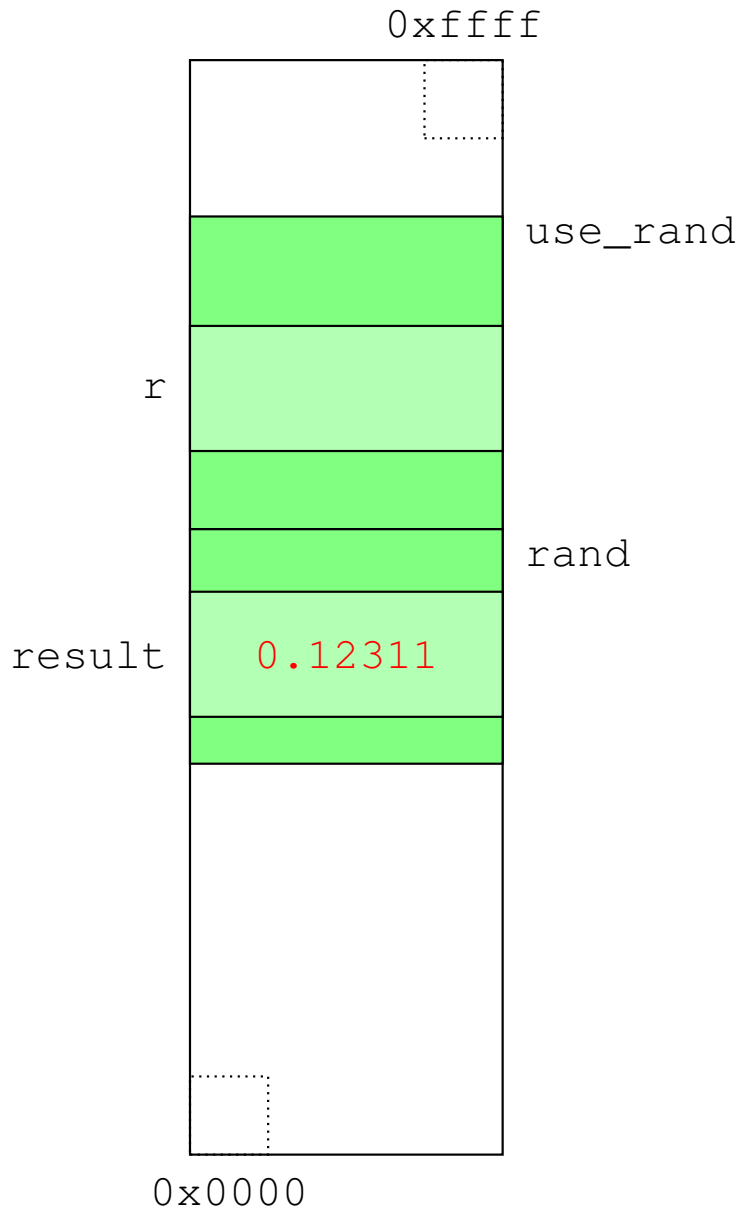


Return a value

```
double rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double r = rand();
  ...
}
```

The return value is copied (C++98) or moved (C++11 onwards) into the destination

- the copy/move can be optimized away

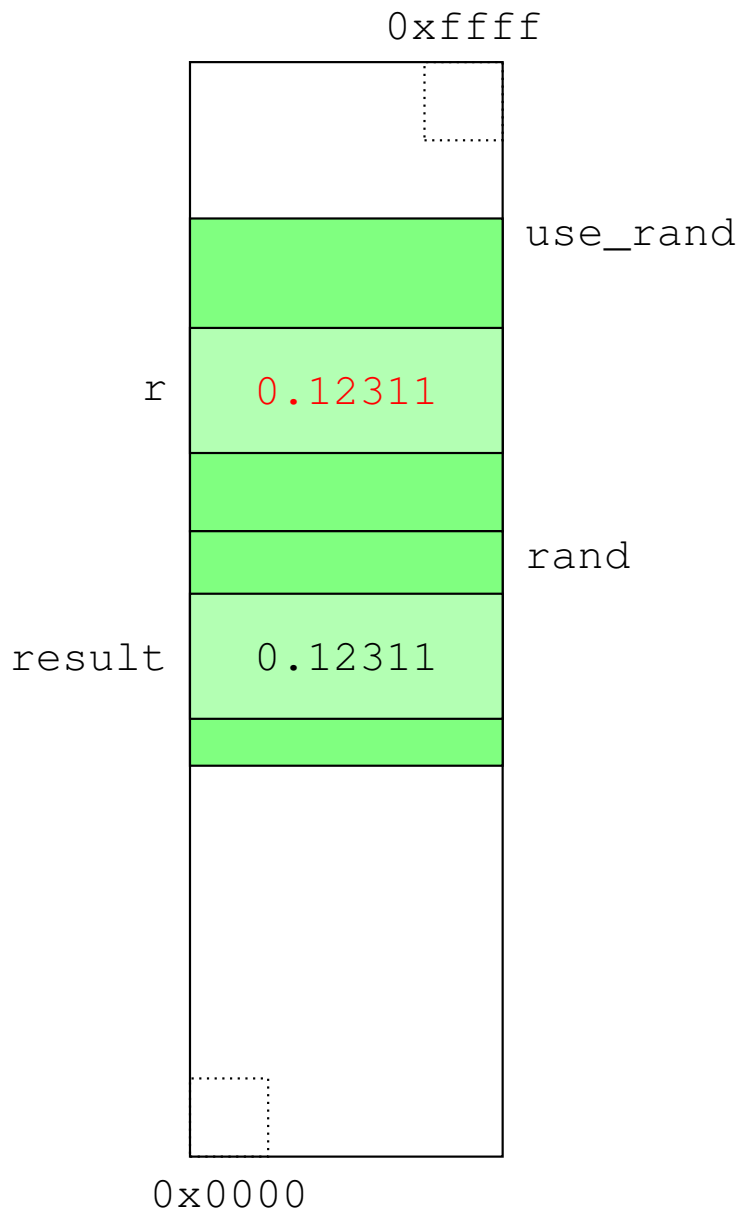


Return a value

```
double rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double r = rand();
  ...
}
```

The return value is copied (C++98) or moved (C++11 onwards) into the destination

- the copy/move can be optimized away

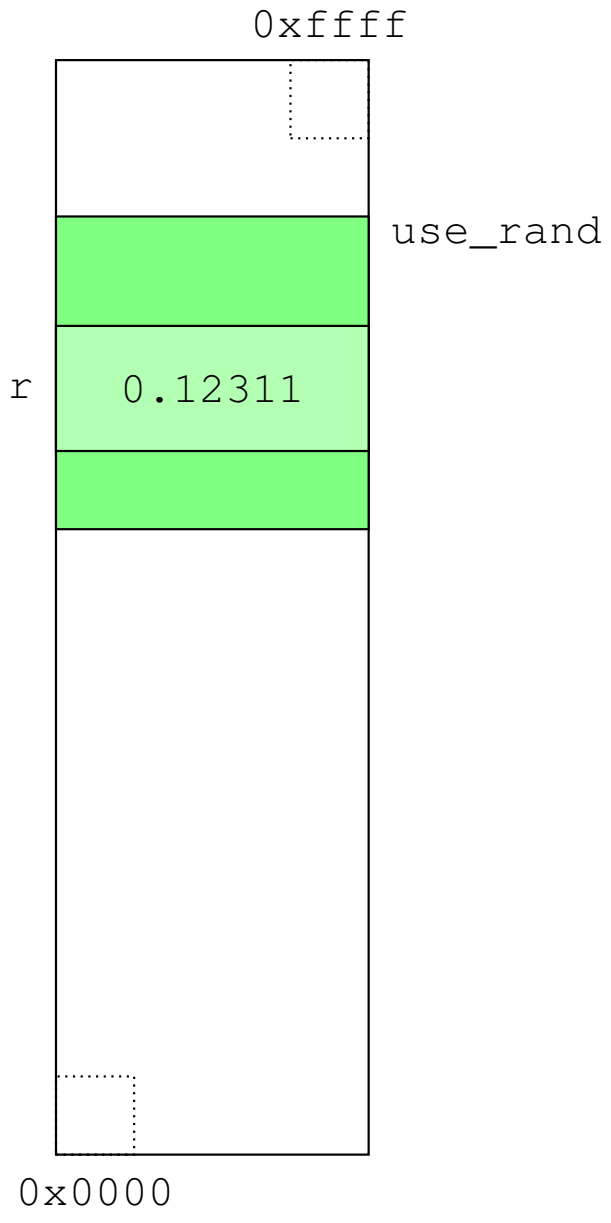


Return a value

```
double rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double r = rand();
  ...
}
```

The return value is copied (C++98) or moved (C++11 onwards) into the destination

- the copy/move can be optimized away

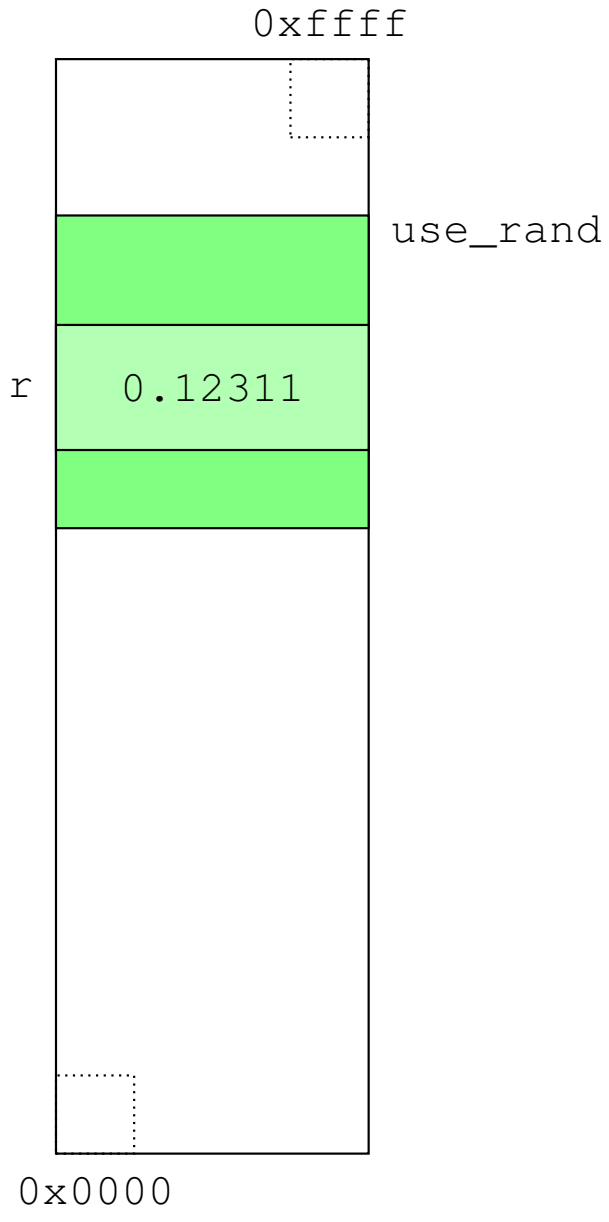


Return a value

```
double rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double r = rand();
  ...
}
```

The return value is copied (C++98) or moved (C++11 onwards) into the destination

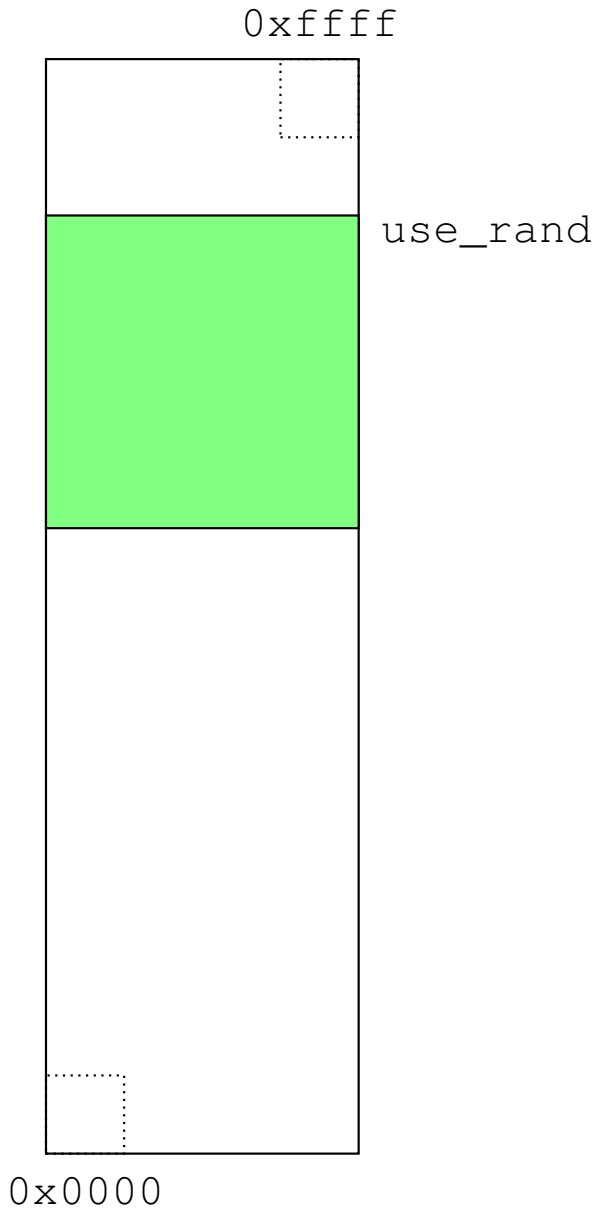
- the copy/move can be optimized away



Return a pointer

Do not return a pointer to a function-local object

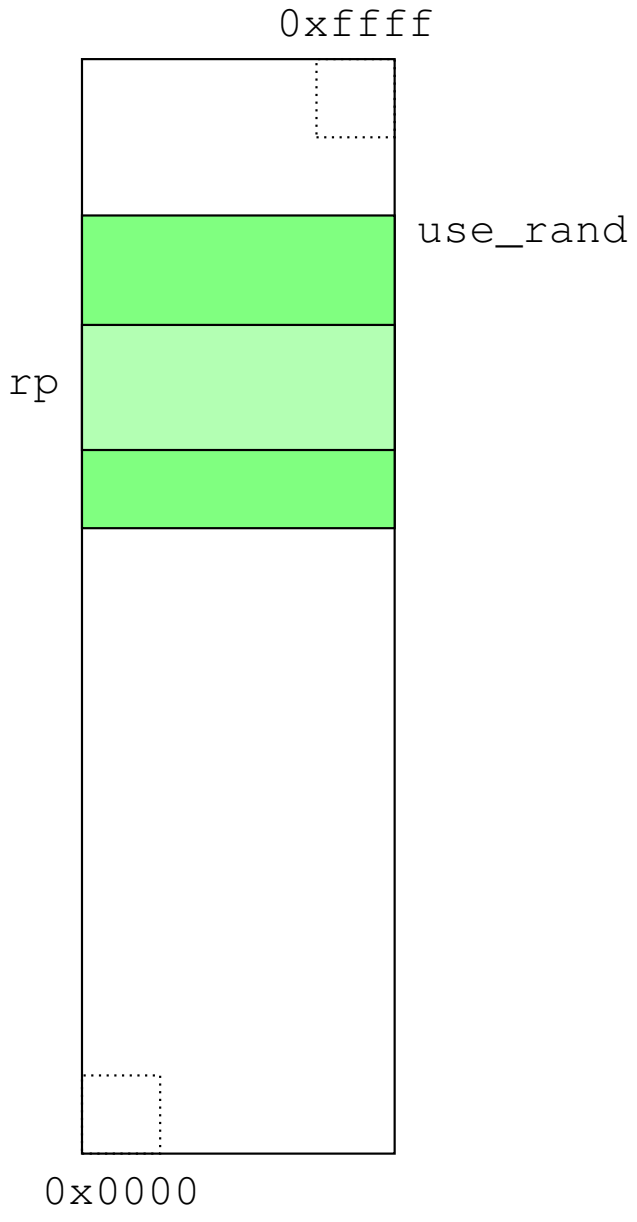
```
double* rand() {
  auto result = 0.;
  ...
  return &result;
}

void use_rand() {
  ...
  double* rp = rand();
  ... *rp ... // ops
}
```



Return a pointer

Do not return a pointer to a function-local object

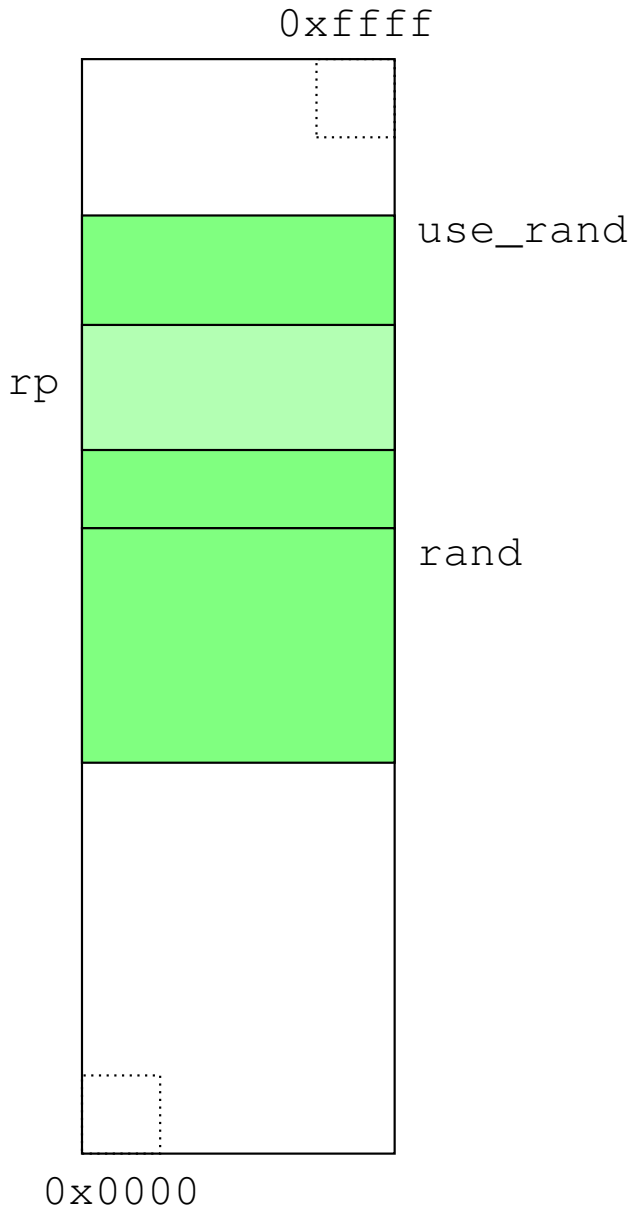
```
double* rand() {
  auto result = 0.;
  ...
  return &result;
}

void use_rand() {
  ...
  double* rp = rand();
  ... *rp ... // ops
}
```



Return a pointer

Do not return a pointer to a function-local object

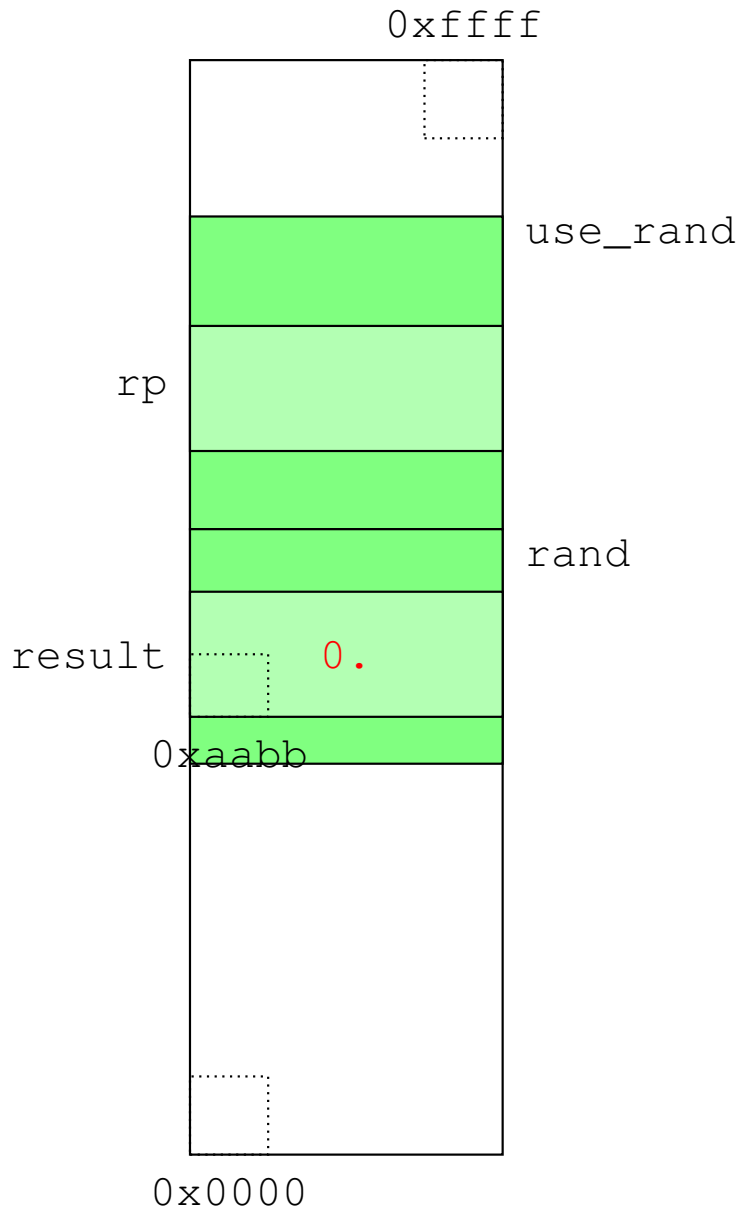
```
double* rand() {
  auto result = 0.;
  ...
  return &result;
}

void use_rand() {
  ...
  double* rp = rand();
  ... *rp ... // ops
}
```



Return a pointer

Do not return a pointer to a function-local object

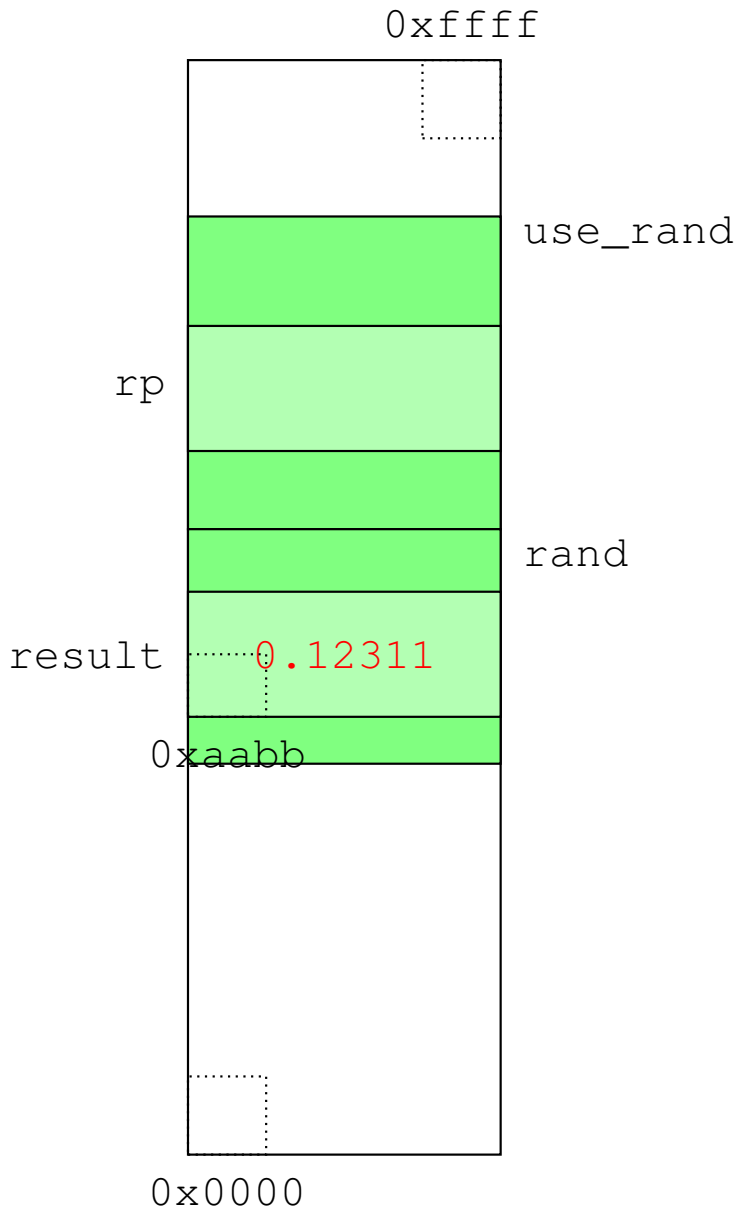
```
double* rand() {
  auto result = 0.;
  ...
  return &result;
}

void use_rand() {
  ...
  double* rp = rand();
  ... *rp ... // ops
}
```



Return a pointer

Do not return a pointer to a function-local object

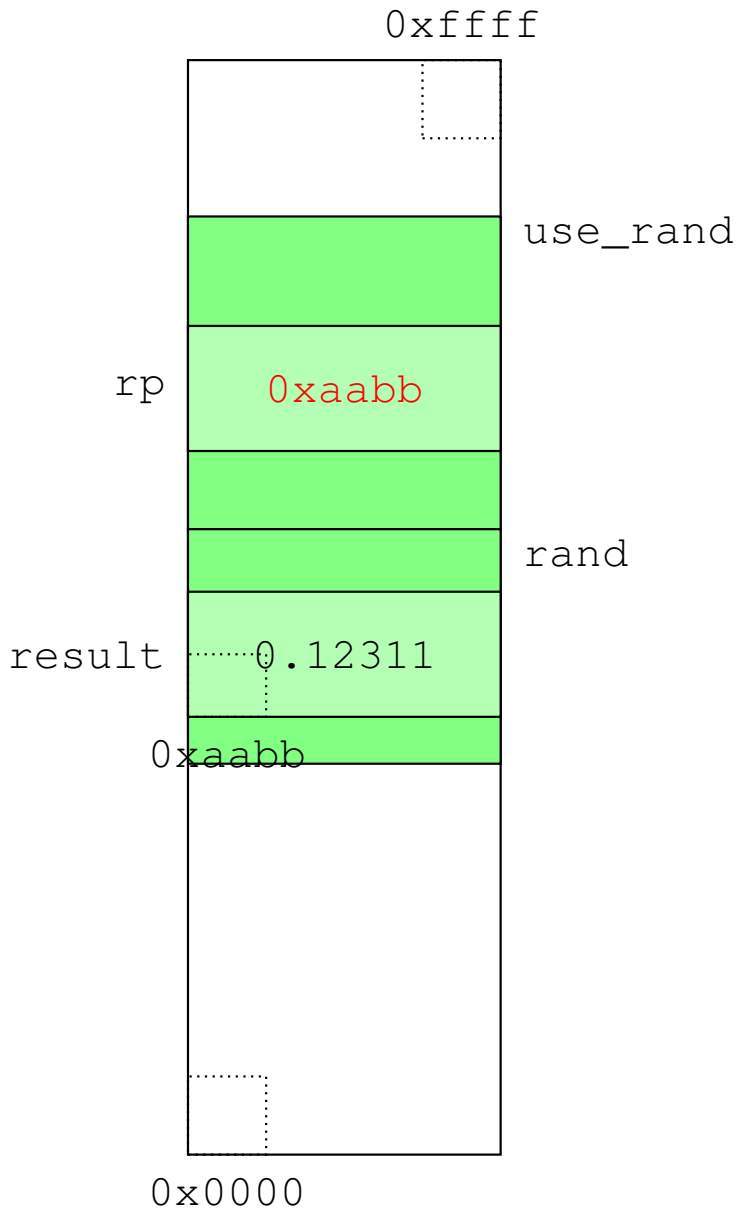
```
double* rand() {
  auto result = 0.;
  ...
  return &result;
}

void use_rand() {
  ...
  double* rp = rand();
  ... *rp ... // ops
}
```



Return a pointer

Do not return a pointer to a function-local object

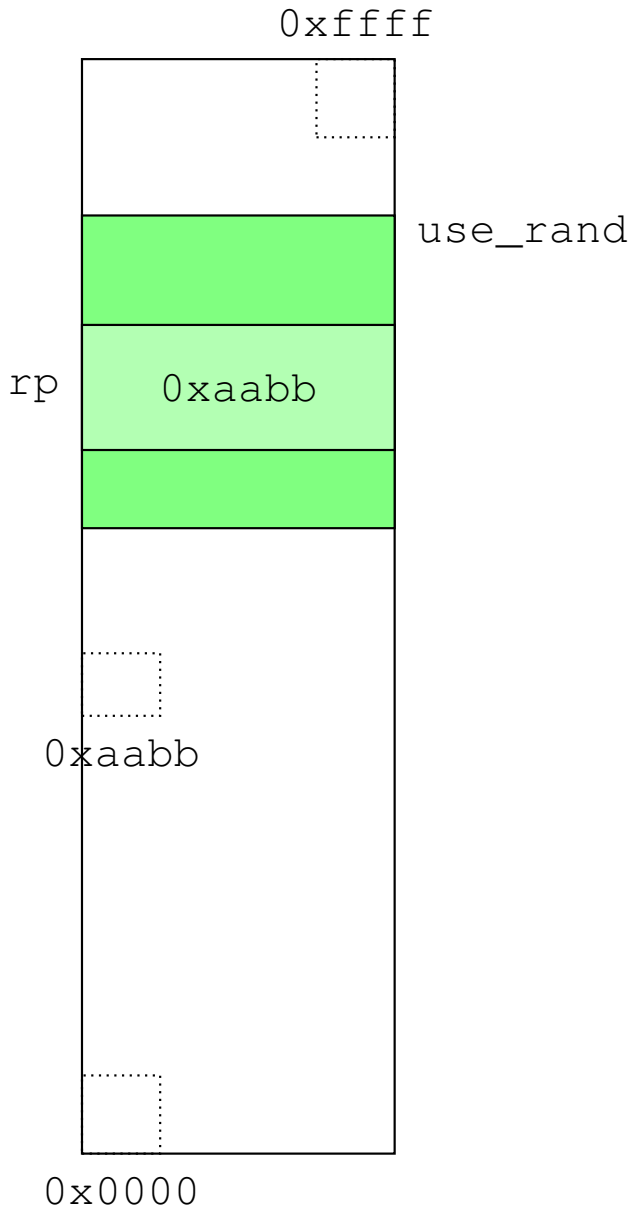
```
double* rand() {
  auto result = 0.;
  ...
  return &result;
}

void use_rand() {
  ...
  double* rp = rand();
  ... *rp ... // ops
}
```



Return a pointer

Do not return a pointer to a function-local object

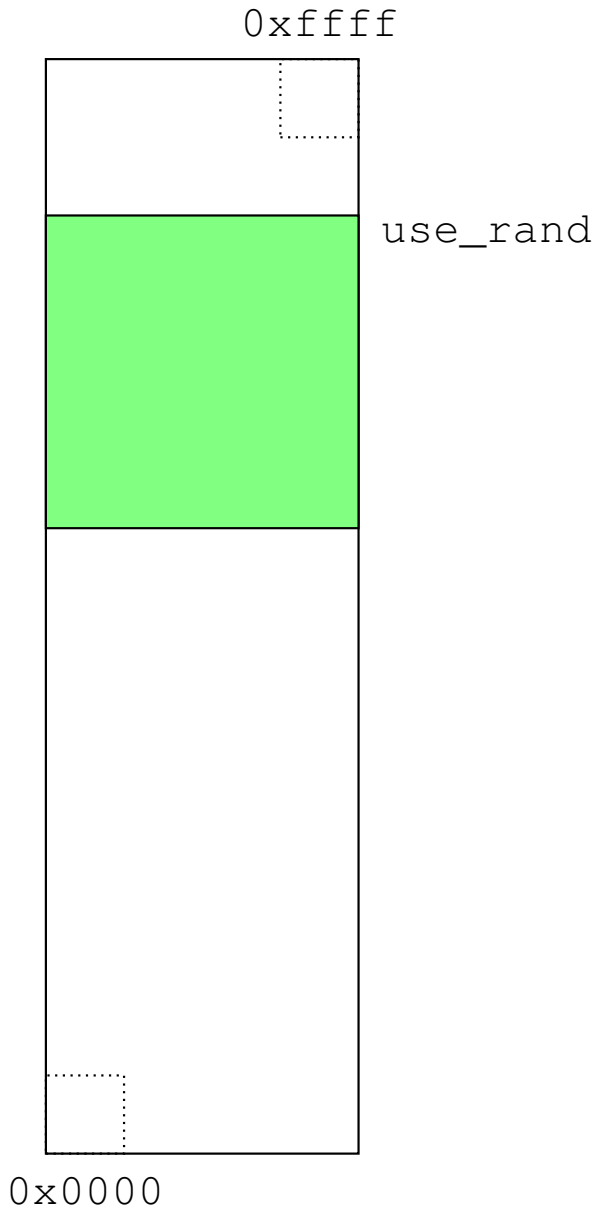
```
double* rand() {
  auto result = 0.;
  ...
  return &result;
}

void use_rand() {
  ...
  double* rp = rand();
  ... *rp ... // ops
}
```



Return a reference

Do not return a reference to a function-local object

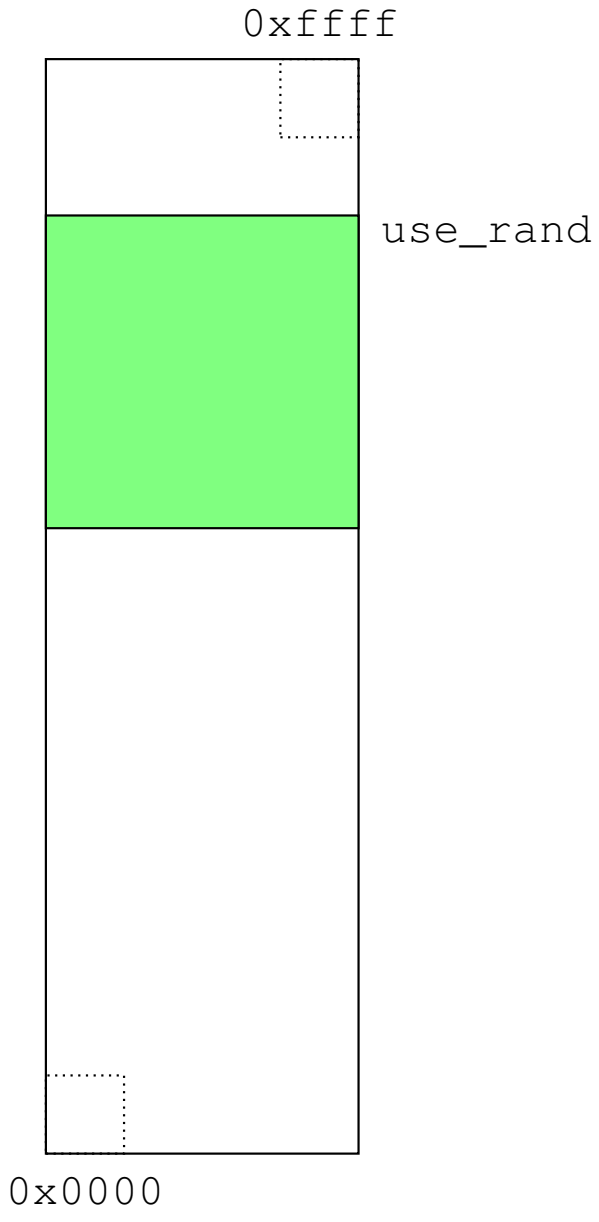
```
double& rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double& r = rand();
  ... r ... // ops
}
```



Return a reference

Do not return a reference to a function-local object

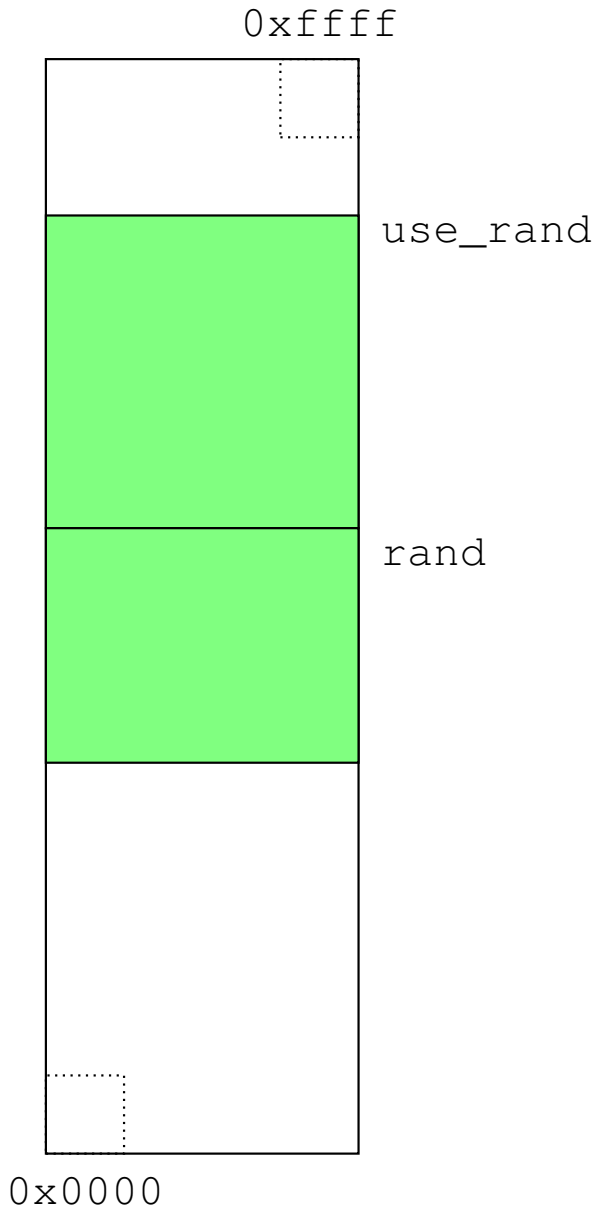
```
double& rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double& r = rand();
  ... r ... // ops
}
```



Return a reference

Do not return a reference to a function-local object

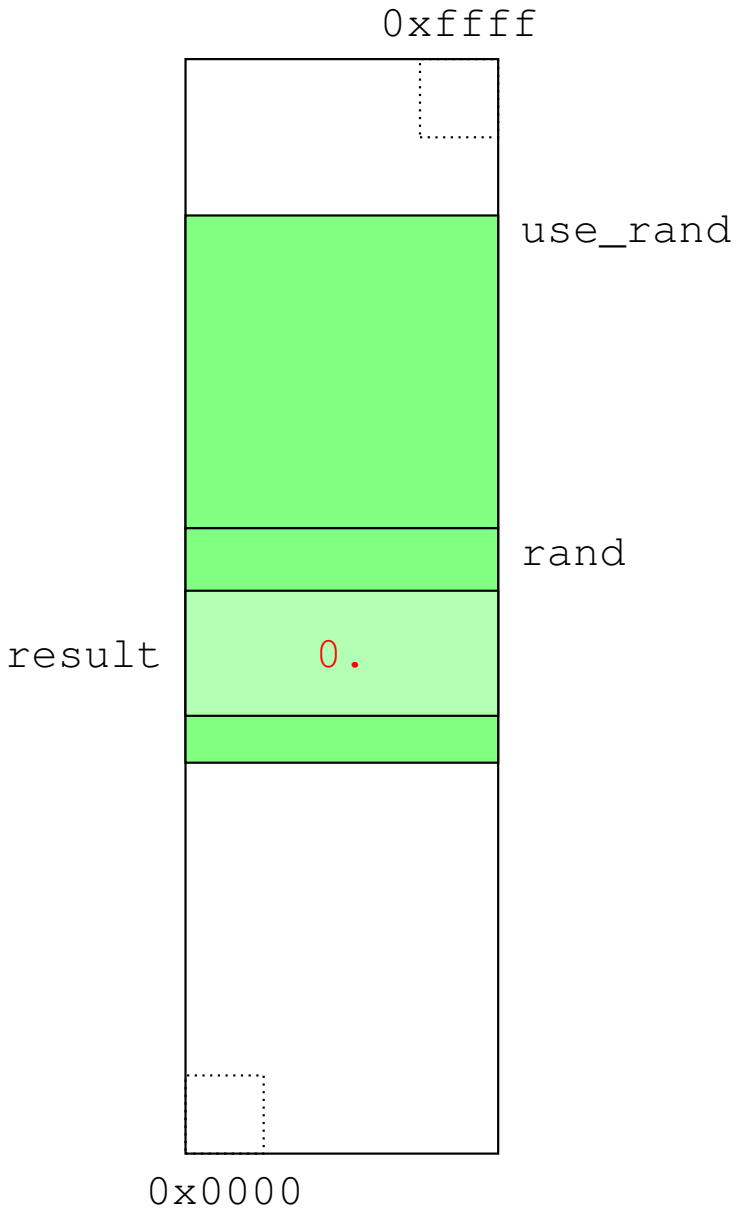
```
double& rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double& r = rand();
  ... r ... // ops
}
```



Return a reference

Do not return a reference to a function-local object

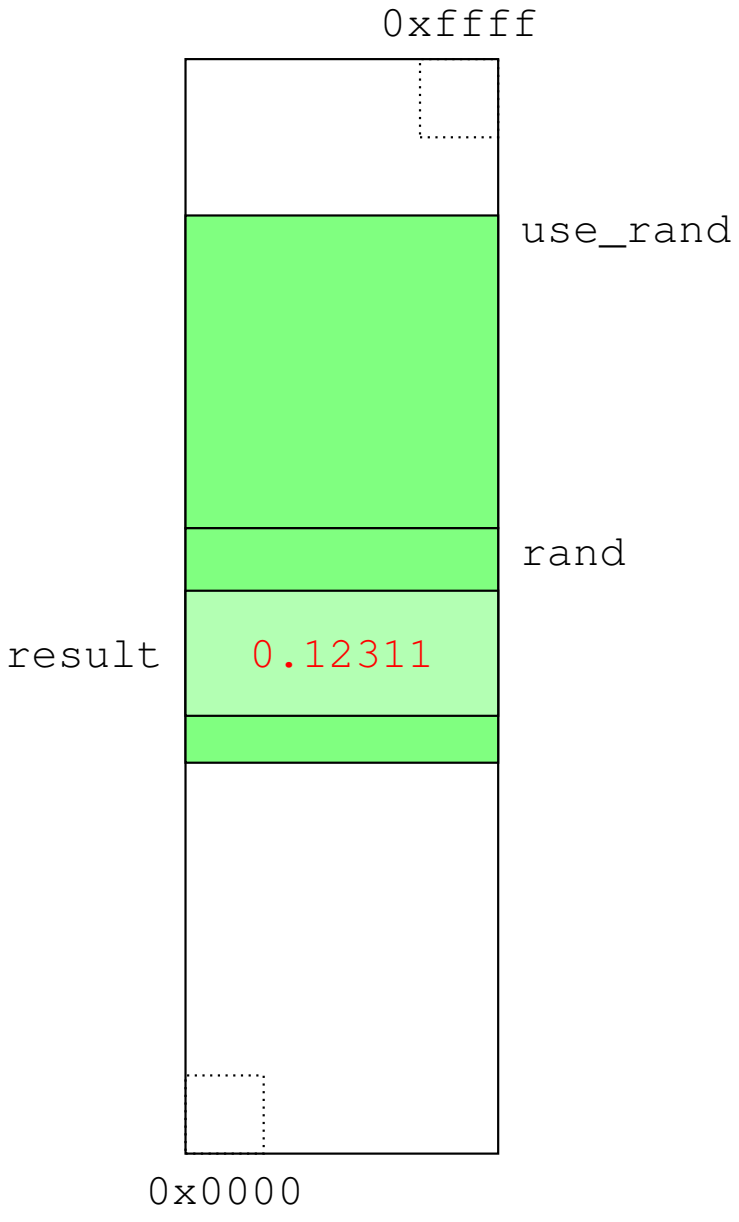
```
double& rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double& r = rand();
  ... r ... // ops
}
```



Return a reference

Do not return a reference to a function-local object

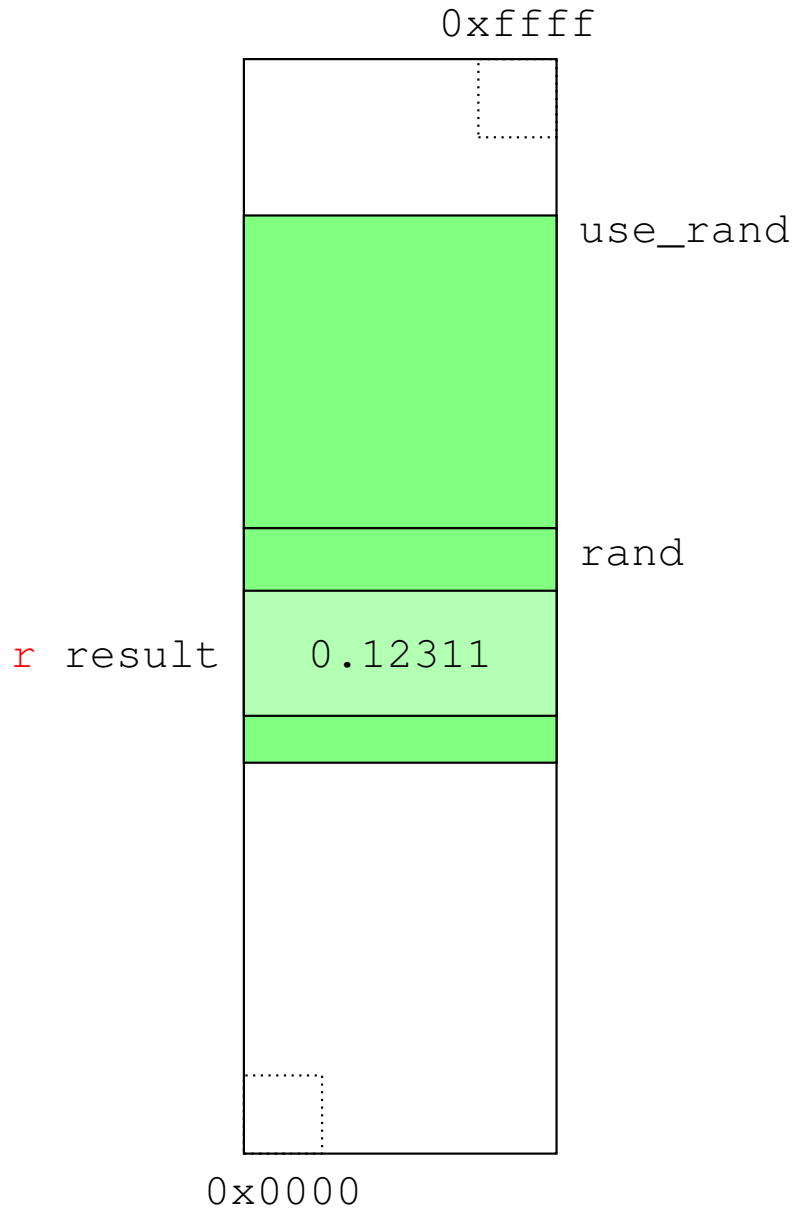
```
double& rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double& r = rand();
  ... r ... // ops
}
```



Return a reference

Do not return a reference to a function-local object

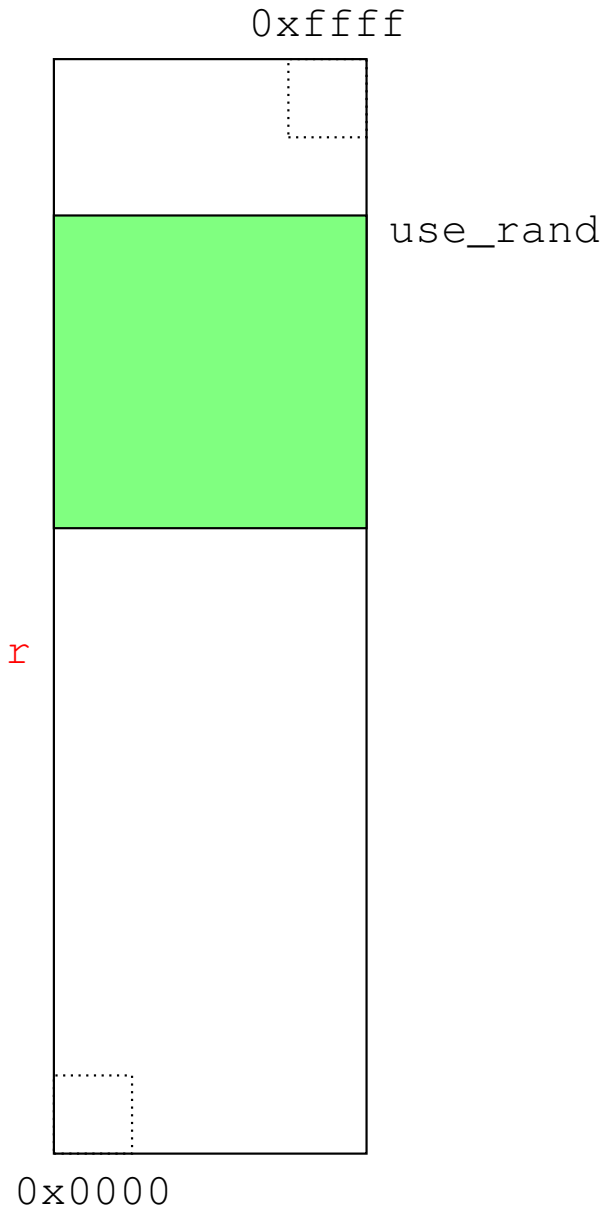
```
double& rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double& r = rand();
  ... r ... // ops
}
```



Return a reference

Do not return a reference to a function-local object

```
double& rand() {
  auto result = 0.;
  ...
  return result;
}

void use_rand() {
  ...
  double& r = rand();
  ... r ... // ops
}
```



Function declaration vs definition

```
void print(double d); // declaration; just the signature
double rand();        // idem

void test_rand()
{
   print(rand());   // call
   ...
}
```

```
// possibly in another file (translation unit)
void print(double d) { ... } // definition; also the body
double rand() { ... }        // idem
```

- For a call, the compiler just needs to know the declaration (signature) of a function
- The linker will then put everything together

if then else

Selection statement to choose one of two flows of control depending on a boolean condition

- `if (` *condition* `)` *statement*
- `if (` *condition* `)` *statement* `else` *statement*

```
int abs(int i) {
  if (i < 0) { i = -i; }
  return i;
}
```

```
int abs(int i) {
  if (i < 0) { return -i; }
  else { return i; }
}
```

condition can be any expression convertible to bool

```
unsigned fact(unsigned n) {
  if (n) { // n != 0
    // recursive call
    return n * fact(n - 1);
  } else { return 1; }
}
```

```
void foo(double* p) {
  if (p) { // p != nullptr
    print(*p);
  }
}
```

if then else

Selection statement to choose one of two flows of control depending on a boolean condition

- `if` (*condition*) *statement*
- `if` (*condition*) *statement* `else` *statement*

```
int abs(int i) {
  if (i < 0) { i = -i; }
  return i;
}
```

```
int abs(int i) {
  if (i < 0) { return -i; }
  else { return i; }
}
```

condition can be any expression convertible to bool

```
unsigned fact(unsigned n) {
  if (n) { // n != 0
    // recursive call
    return n * fact(n - 1);
  } else { return 1; }
}
```

```
void foo(double* p) {
  if (p) { // p != nullptr
    print(*p);
  }
}
```

if then else

Selection statement to choose one of two flows of control depending on a boolean condition

- `if` `(` *condition* `)` *statement*
- `if` `(` *condition* `)` *statement* `else` *statement*

```
int abs(int i) {
  if (i < 0) { i = -i; }
  return i;
}
```

```
int abs(int i) {
  if (i < 0) { return -i; }
  else { return i; }
}
```

condition can be any expression convertible to bool

```
unsigned fact(unsigned n) {
  if (n) { // n != 0
    // recursive call
    return n * fact(n - 1);
  } else { return 1; }
}
```

```
void foo(double* p) {
  if (p) { // p != nullptr
    print(*p);
  }
}
```

if then else

Selection statement to choose one of two flows of control depending on a boolean condition

- `if` (*condition*) *statement*
- `if` (*condition*) *statement* `else` *statement*

```
int abs(int i) {
  if (i < 0) { i = -i; }
  return i;
}
```

```
int abs(int i) {
  if (i < 0) { return -i; }
  else { return i; }
}
```

condition can be any expression convertible to bool

```
unsigned fact(unsigned n) {
  if (n) { // n != 0
    // recursive call
    return n * fact(n - 1);
  } else { return 1; }
}
```

```
void foo(double* p) {
  if (p) { // p != nullptr
    print(*p);
  }
}
```

while

`while` `(` *condition* `)` *statement*

- Execute repeatedly *statement* until *condition* becomes false
- *condition* is evaluated at the beginning of each iteration

```
bool is_prime(unsigned n)
{
  if (n < 3) { return true; }
  auto div = 2;
  while (div <= sqrt(n)) {
    if (!(n % div)) {
      return false;
    }
    ++div;
  }
  return true;
}
```

```
bool is_prime(unsigned n)
{
  if (n < 3) { return true; }
  auto maybe_prime = true;
  auto div = 2;
  while (div <= sqrt(n)
         && maybe_prime) {
    maybe_prime = n % div;
    ++div;
  }
  return maybe_prime;
}
```

while

`while` (*condition*) *statement*

- Execute repeatedly *statement* until *condition* becomes false
- *condition* is evaluated at the beginning of each iteration

```
bool is_prime(unsigned n)
{
  if (n < 3) { return true; }
  auto div = 2;
  while (div <= sqrt(n)) {
    if (!(n % div)) {
      return false;
    }
    ++div;
  }
  return true;
}
```

```
bool is_prime(unsigned n)
{
  if (n < 3) { return true; }
  auto maybe_prime = true;
  auto div = 2;
  while (div <= sqrt(n)
         && maybe_prime) {
    maybe_prime = n % div;
    ++div;
  }
  return maybe_prime;
}
```

while

`while` (*condition*) *statement*

- Execute repeatedly *statement* until *condition* becomes false
- *condition* is evaluated at the beginning of each iteration

```
bool is_prime(unsigned n)
{
  if (n < 3) { return true; }
  auto div = 2;
  while (div <= sqrt(n)) {
    if (!(n % div)) {
      return false;
    }
    ++div;
  }
  return true;
}
```

```
bool is_prime(unsigned n)
{
  if (n < 3) { return true; }
  auto maybe_prime = true;
  auto div = 2;
  while (div <= sqrt(n)
         && maybe_prime) {
    maybe_prime = n % div;
    ++div;
  }
  return maybe_prime;
}
```

for

`for` ($init_{opt}$ `;` $condition_{opt}$ `;` $expression_{opt}$) *statement* (simplified)

- Execute repeatedly *statement* until *condition* becomes false
- *init* is executed before entering the loop
- *condition* is evaluated at the beginning of each iteration
- *expression* is evaluated at the end of each iteration

The following are more or less equivalent

```
bool is_prime(unsigned n) {
  if (n < 3) { return true; }
  auto maybe_prime = true;
  auto div = 2;
  while (div <= sqrt(n)
         && maybe_prime) {
    maybe_prime = n % div;
    ++div;
  }
  return maybe_prime;
}
```

```
bool is_prime(unsigned n) {
  if (n < 3) { return true; }
  auto maybe_prime = true;
  for (auto div = 2;
       div <= sqrt(n)
       && maybe_prime;
       ++div) {
    maybe_prime = n % div;
  }
  return maybe_prime;
}
```

for

`for` ($init_{opt}$ `;` $condition_{opt}$ `;` $expression_{opt}$) *statement* (simplified)

- Execute repeatedly *statement* until *condition* becomes false
- *init* is executed before entering the loop
- *condition* is evaluated at the beginning of each iteration
- *expression* is evaluated at the end of each iteration

The following are more or less equivalent

```
bool is_prime(unsigned n) {
  if (n < 3) { return true; }
  auto maybe_prime = true;
  auto div = 2;
  while (div <= sqrt(n)
         && maybe_prime) {
    maybe_prime = n % div;
    ++div;
  }
  return maybe_prime;
}
```

```
bool is_prime(unsigned n) {
  if (n < 3) { return true; }
  auto maybe_prime = true;
  for (auto div = 2;
       div <= sqrt(n)
       && maybe_prime;
       ++div) {
    maybe_prime = n % div;
  }
  return maybe_prime;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1; // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return                    ;
}

double norm2(Complex const& c) {
  return                          ;
}

double norm2(Complex const* c) {

  return

                              ;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return                    ;
}

double norm2(Complex const& c) {
  return                          ;
}

double norm2(Complex const* c) {

  return

                              ;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r; // data member
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return                    ;
}

double norm2(Complex const& c) {
  return                    ;
}

double norm2(Complex const* c) {

  return

                            ;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return                    ;
}

double norm2(Complex const& c) {
  return                    ;
}

double norm2(Complex const* c) {

  return

                            ;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const* c) {
  assert(c != nullptr);
  return
    (*c).r *                    ;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const* c) {
  assert(c != nullptr);
  return
    (*c).r *                    ;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const* c) {
  assert(c != nullptr);
  return
    (*c).r *                    ;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const* c) {
  assert(c != nullptr);
  return
    (*c).r * (*c).r + (*c).i * (*c).i;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const* c) {
  assert(c != nullptr);
  return
    (*c).r *                    ;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const* c) {
  assert(c != nullptr);
  return
    (*c).r *                    ;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const* c) {
  assert(c != nullptr);
  return
    (*c).r *                    ;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const* c) {
  assert(c != nullptr);
  return
    (*c).r * c->r + c->i * c->i;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const* c) {
  assert(c != nullptr);
  return
    (*c).r * c->r + c->i * c->i;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const* c) {
  assert(c != nullptr);
  return
    (*c).r * c->r + c->i * c->i;
}
```

`class` and `struct`

- `class` and `struct` are the primary mechanisms to define new types on top of fundamental types
- They are almost synonyms

```
struct Complex {
  double r;
  double i;
};

double norm2(Complex);
Complex sqrt(Complex);

Complex c1;
norm(c1);
Complex c2 = sqrt(c1);

Complex& cr = c1;  // reference
Complex* cp = &c1; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const* c) {
  assert(c != nullptr);
  return
    (*c).r * c->r + c->i * c->i;
}
```

Construction

- A special function, called constructor, can be associated to a class to initialize an object of that type when it is created
- The constructor is named after the class name

```
struct Complex {
  double r;
  double i;
  Complex(double x, double y) // no return type
      : r{x}, i{y} // member initialization list
  {
    // nothing else to do
  }
};

Complex c{1., 2.};
norm2(c);

// alternative, almost equivalent syntaxes
Complex c = {1., 2.};
Complex c(1., 2.);
auto c = Complex{1., 2.};

Complex c{1.}; // what about constructing from a single double?
```

Construction

- A special function, called constructor, can be associated to a class to initialize an object of that type when it is created
- The constructor is named after the class name

```
struct Complex {
  double r;
  double i;
  Complex(double x, double y) // no return type
      : r{x}, i{y} // member initialization list
  {
   // nothing else to do
  }
};

Complex c{1., 2.};
norm2(c);

// alternative, almost equivalent syntaxes
Complex c = {1., 2.};
Complex c(1., 2.);
auto c = Complex{1., 2.};

Complex c{1.}; // what about constructing from a single double?
```

Construction

- A special function, called constructor, can be associated to a class to initialize an object of that type when it is created
- The constructor is named after the class name

```
struct Complex {
  double r;
  double i;
  Complex(double x, double y) // no return type
      : r{x}, i{y} // member initialization list
  {
   // nothing else to do
  }
};

Complex c{1., 2.};
norm2(c);

// alternative, almost equivalent syntaxes
Complex c = {1., 2.};
Complex c(1., 2.);
auto c = Complex{1., 2.};

Complex c{1.}; // what about constructing from a single double?
```

Construction

- A special function, called constructor, can be associated to a class to initialize an object of that type when it is created
- The constructor is named after the class name

```
struct Complex {
  double r;
  double i;
  Complex(double x, double y) // no return type
      : r{x}, i{y} // member initialization list
  {
   // nothing else to do
  }
};

Complex c{1., 2.};
norm2(c);

// alternative, almost equivalent syntaxes
Complex c = {1., 2.};
Complex c(1., 2.);
auto c = Complex{1., 2.};

Complex c{1.}; // what about constructing from a single double?
```

Construction

- A special function, called constructor, can be associated to a class to initialize an object of that type when it is created
- The constructor is named after the class name

```
struct Complex {
  double r;
  double i;
  Complex(double x, double y) // no return type
      : r{x}, i{y} // member initialization list
  {
   // nothing else to do
  }
};

Complex c{1., 2.};
norm2(c);

// alternative, almost equivalent syntaxes
Complex c = {1., 2.};
Complex c(1., 2.);
auto c = Complex{1., 2.};

Complex c{1.}; // what about constructing from a single double?
```

Construction

- A special function, called constructor, can be associated to a class to initialize an object of that type when it is created
- The constructor is named after the class name

```
struct Complex {
  double r;
  double i;
  Complex(double x, double y) // no return type
      : r{x}, i{y} // member initialization list
  {
   // nothing else to do
  }
};

Complex c{1., 2.};
norm2(c);

// alternative, almost equivalent syntaxes
Complex c = {1., 2.};
Complex c(1., 2.);
auto c = Complex{1., 2.};

Complex c{1.}; // what about constructing from a single double?
```

Function overloading

A program can contain multiple functions with the same name

- they must have different parameter lists; the return type is not involved
- for a given call, the compiler will choose the best match, if any, between the given arguments and the available parameter lists
- the matching algorithm is complex and involves implicit and explicit, standard and user-defined conversions

```
unsigned square(int);       // (1)
unsigned square(unsigned); // (2)
double square(double);      // (3)

square(3);                          // calls (1)
square(3U);                         // calls (2)
square(3L);                         // ambiguous
square(static_cast<int>(3L)); // calls (1)
square(1.);                         // calls (3)
square(1.F);                        // calls (3)
```

Function overloading

A program can contain multiple functions with the same name

- they must have different parameter lists; the return type is not involved
- for a given call, the compiler will choose the best match, if any, between the given arguments and the available parameter lists
- the matching algorithm is complex and involves implicit and explicit, standard and user-defined conversions

```
unsigned square(int);       // (1)
unsigned square(unsigned); // (2)
double square(double);      // (3)

square(3);                          // calls (1)
square(3U);                         // calls (2)
square(3L);                         // ambiguous
square(static_cast<int>(3L)); // calls (1)
square(1.);                         // calls (3)
square(1.F);                        // calls (3)
```

Function overloading

A program can contain multiple functions with the same name

- they must have different parameter lists; the return type is not involved
- for a given call, the compiler will choose the best match, if any, between the given arguments and the available parameter lists
- the matching algorithm is complex and involves implicit and explicit, standard and user-defined conversions

```
unsigned square(int);       // (1)
unsigned square(unsigned); // (2)
double square(double);      // (3)

square(3);                          // calls (1)
square(3U);                         // calls (2)
square(3L);                         // ambiguous
square(static_cast<int>(3L)); // calls (1)
square(1.);                         // calls (3)
square(1.F);                        // calls (3)
```

Function overloading

A program can contain multiple functions with the same name

- they must have different parameter lists; the return type is not involved
- for a given call, the compiler will choose the best match, if any, between the given arguments and the available parameter lists
- the matching algorithm is complex and involves implicit and explicit, standard and user-defined conversions

```
unsigned square(int);       // (1)
unsigned square(unsigned); // (2)
double square(double);      // (3)

square(3);                      // calls (1)
square(3U);                     // calls (2)
square(3L);                     // ambiguous
square(static_cast<int>(3L)); // calls (1)
square(1.);                     // calls (3)
square(1.F);                    // calls (3)
```

Construction (cont.)

```
struct Complex {
  double r;
  double i;




};

Complex c1{1., 2.};
Complex c2{1.}; // meaning {1., 0.}
Complex c3; // or {}, but not (); meaning {0., 0.}
```

- A constructor should initialize all the data members
- The default constructor is automatically generated by the compiler if there are no other constructors

Construction (cont.)

```
struct Complex {
  double r;
  double i;
  Complex(double x, double y) : r{x}, i{y} {}
  Complex(double x) : r{x}, i{0.} {}
  Complex() : r{0.}, i{0.} {} // default constructor

};

Complex c1{1., 2.};
Complex c2{1.}; // meaning {1., 0.}
Complex c3; // or {}, but not (); meaning {0., 0.}
```

- A constructor should initialize all the data members
- The default constructor is automatically generated by the compiler if there are no other constructors

Construction (cont.)

```
struct Complex {
  double r;
  double i;
  Complex(double x, double y) : r{x}, i{y} {}
  Complex(double x) : Complex{x, 0.} {} // delegating constructor
  Complex() : Complex{0., 0.} {}

};

Complex c1{1., 2.};
Complex c2{1.}; // meaning {1., 0.}
Complex c3; // or {}, but not (); meaning {0., 0.}
```

- A constructor should initialize all the data members
- The default constructor is automatically generated by the compiler if there are no other constructors

Construction (cont.)

```
struct Complex {
  double r = 0.; // default member initializer
  double i = 0.;
  Complex(double x, double y) : r{x}, i{y} {}
  Complex(double x) : r{x} {}
  Complex() {}

};

Complex c1{1., 2.};
Complex c2{1.}; // meaning {1., 0.}
Complex c3; // or {}, but not (); meaning {0., 0.}
```

- A constructor should initialize all the data members
- The default constructor is automatically generated by the compiler if there are no other constructors

Construction (cont.)

```
struct Complex {
  double r;
  double i;
  Complex(double x = 0., double y = 0.) // default arguments
       : r{x}, i{y} {}


};

Complex c1{1., 2.};
Complex c2{1.}; // meaning {1., 0.}
Complex c3; // or {}, but not (); meaning {0., 0.}
```

- A constructor should initialize all the data members
- The default constructor is automatically generated by the compiler if there are no other constructors

Private representation, public interface

- The internal representation of a class should be considered an implementation detail
- The manipulation of objects should happen through a well-defined function-based interface

```
class Complex {
 private: // representation based on cartesian coordinates
  double r;
  double i;
 public:
  Complex(double x = 0., double y = 0.) : r{x}, i{y} {}
  double real() { return r; }
  double imag() { return i; }
};

double norm2(Complex const& c) {

}
```

Private representation, public interface

- The internal representation of a class should be considered an implementation detail
- The manipulation of objects should happen through a well-defined function-based interface

```
class Complex {
 private: // representation based on cartesian coordinates
  double r;
  double i;
 public:
  Complex(double x = 0., double y = 0.) : r{x}, i{y} {}
  double real() { return r; }
  double imag() { return i; }
};

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}
```

Private representation, public interface

- The internal representation of a class should be considered an implementation detail
- The manipulation of objects should happen through a well-defined function-based interface

```
class Complex {
 private: // representation based on cartesian coordinates
  double r;
  double i;
 public:
  Complex(double x = 0., double y = 0.) : r{x}, i{y} {}
  double real() { return r; }
  double imag() { return i; }
};

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i; // error
}
```

Private representation, public interface

- The internal representation of a class should be considered an implementation detail
- The manipulation of objects should happen through a well-defined function-based interface

```
class Complex {
 private: // representation based on cartesian coordinates
  double r;
  double i;
 public:
  Complex(double x = 0., double y = 0.) : r{x}, i{y} {}
  double real() { return r; } // member function (aka method)
  double imag() { return i; }
};

double norm2(Complex const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
```

Private representation, public interface

- The internal representation of a class should be considered an implementation detail
- The manipulation of objects should happen through a well-defined function-based interface

```
class Complex {
 private: // representation based on cartesian coordinates
  double r;
  double i;
 public:
  Complex(double x = 0., double y = 0.) : r{x}, i{y} {}
  double real() { return r; } // could modify data members
  double imag() { return i; }
};

double norm2(Complex const& c) {
  return c.real() * c.real() + c.imag() * c.imag(); // error
}
```

Private representation, public interface

- The internal representation of a class should be considered an implementation detail
- The manipulation of objects should happen through a well-defined function-based interface

```
class Complex {
 private: // representation based on cartesian coordinates
  double r;
  double i;
 public:
  Complex(double x = 0., double y = 0.) : r{x}, i{y} {}
  double real() const { return r; }
  double imag() const { return i; }
};

double norm2(Complex const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
```

Member vs free function

```
class Complex {
 public:
  double norm2() const { // member function
    return r * r + i * i;
  }
  ...
};

double norm2(Complex const& c) { // free function
  return c.real() * c.real() + c.imag() * c.imag();
}

Complex c;
c.norm2(); // call the member function
norm2(c);  // call the free function
```

- The public interface of a class should be minimal
- Implement a free function if you can, a member function if you must

Member vs free function

```
class Complex {
 public:
  double norm2() const { // member function
    return r * r + i * i;
  }
  ...
};

double norm2(Complex const& c) { // free function
  return c.real() * c.real() + c.imag() * c.imag();
}

Complex c;
c.norm2(); // call the member function
norm2(c);  // call the free function
```

- The public interface of a class should be minimal
- Implement a free function if you can, a member function if you must

Member vs free function

```
class Complex {
 public:
  double norm2() const { // member function
    return r * r + i * i;
  }
  ...
};

double norm2(Complex const& c) { // free function
  return c.real() * c.real() + c.imag() * c.imag();
}

Complex c;
c.norm2(); // call the member function
norm2(c);  // call the free function
```

- The public interface of a class should be minimal
- Implement a free function if you can, a member function if you must

Class scope

The (potential) scope of a name declared in a class begins at the point of declaration and includes the rest of the class body, all function bodies, all nested classes, . . .

Operator overloading

```
class Complex {
  ...
};

bool operator==(Complex const& lhs, Complex const& rhs) {
  return lhs.real() == rhs.real() && lhs.imag() == rhs.imag();
}

c2 = c1 // generated by the compiler
c1 == c2
c1 += c2
c1 + c2
2. * c1
z = z * z + c
...
```

Operator overloading

```
class Complex {
  ...
};

bool operator==(Complex const& lhs, Complex const& rhs) {
  return lhs.real() == rhs.real() && lhs.imag() == rhs.imag();
}

c2 = c1 // generated by the compiler
c1 == c2
c1 += c2
c1 + c2
2. * c1
z = z * z + c
...
```

Operator overloading

```
class Complex {
  ...
};

bool operator==(Complex const& lhs, Complex const& rhs) {
  return lhs.real() == rhs.real() && lhs.imag() == rhs.imag();
}

c2 = c1 // generated by the compiler
c1 == c2
c1 += c2
c1 + c2
2. * c1
z = z * z + c
...
```

Operator overloading

```
class Complex {
  ...
};

bool operator==(Complex const& lhs, Complex const& rhs) {
  return lhs.real() == rhs.real() && lhs.imag() == rhs.imag();
}

c2 = c1 // generated by the compiler
c1 == c2
c1 += c2
c1 + c2
2. * c1
z = z * z + c
...
```

this

Within the body of a member function of a class `T`, `this`

- is a pointer of type `T*`
- points to the object for which the function was called

```
struct Foo {
  void bar() {
    ... this ...; // address of beta (i.e. &beta)
  }
};

Foo alfa;
Foo beta;
alfa.bar();
beta.bar();
```

this

Within the body of a member function of a class `T`, `this`

- is a pointer of type `T*`
- points to the object for which the function was called

```
struct Foo {
  void bar() {
    ... this ...; // address of beta (i.e. &beta)
  }
};

Foo alfa;
Foo beta;
alfa.bar();
beta.bar();
```

this

Within the body of a member function of a class `T`, `this`

- is a pointer of type `T*`
- points to the object for which the function was called

```
struct Foo {
  void bar() {
    ... this ...; // address of beta (i.e. &beta)
  }
};

Foo alfa;
Foo beta;
alfa.bar();
beta.bar();
```

this

Within the body of a member function of a class `T`, `this`

- is a pointer of type `T*`
- points to the object for which the function was called

```
struct Foo {
  void bar() {
    ... this ...; // address of alfa (i.e. &alfa)
  }
};

Foo alfa;
Foo beta;
alfa.bar();
beta.bar();
```

this

Within the body of a member function of a class `T`, `this`

- is a pointer of type `T*`
- points to the object for which the function was called

```
struct Foo {
  void bar() {
    ... this ...; // address of beta (i.e. &beta)
  }
};

Foo alfa;
Foo beta;
alfa.bar();
beta.bar();
```

Operator overloading (cont.)

```
class Complex {
  ...
  Complex& operator+=(Complex const& rhs) {
    r += rhs.r;
    i += rhs.i;
    return *this;
  }
};

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs; // compiler-generated special constructor
  result += rhs;
  return result;
}

Complex operator*(double s, Complex const& rhs) {
  return Complex{s * rhs.real(), s * rhs.imag()};
}

c1 += c2
c1 + c2
2. * c1
```

Operator overloading (cont.)

```
class Complex {
  ...
  Complex& operator+=(Complex const& rhs) {
    r += rhs.r;
    i += rhs.i;
    return *this;
  }
};

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs; // compiler-generated special constructor
  result += rhs;
  return result;
}

Complex operator*(double s, Complex const& rhs) {
  return Complex{s * rhs.real(), s * rhs.imag()};
}

c1 += c2
c1 + c2
2. * c1
```

Operator overloading (cont.)

```
class Complex {
  ...
  Complex& operator+=(Complex const& rhs) { // member function
    r += rhs.r;
    i += rhs.i;
    return *this;
  }
};

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs; // compiler-generated special constructor
  result += rhs;
  return result;
}

Complex operator*(double s, Complex const& rhs) {
  return Complex{s * rhs.real(), s * rhs.imag()};
}

c1 += c2 // equivalent to c1.operator+=(c2)
c1 + c2
2. * c1
```

Operator overloading (cont.)

```
class Complex {
  ...
  Complex& operator+=(Complex const& rhs) {
    r += rhs.r;
    i += rhs.i;
    return *this;
  }
};

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs; // compiler-generated special constructor
  result += rhs;
  return result;
}

Complex operator*(double s, Complex const& rhs) {
  return Complex{s * rhs.real(), s * rhs.imag()};
}

c1 += c2 // (c1 += c2).real() should work
c1 + c2
2. * c1
```

Operator overloading (cont.)

```
class Complex {
  ...
  Complex& operator+=(Complex const& rhs) {
    r += rhs.r;
    i += rhs.i;
    return *this;
  }
};

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs; // compiler-generated special constructor
  result += rhs;
  return result;
}

Complex operator*(double s, Complex const& rhs) {
  return Complex{s * rhs.real(), s * rhs.imag()};
}

c1 += c2 // (c1 += c2).real() should work
c1 + c2
2. * c1
```

Operator overloading (cont.)

```
class Complex {
  ...
  Complex& operator+=(Complex const& rhs) {
    r += rhs.r; // access control is per class not per object
    i += rhs.i;
    return *this;
  }
};

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs; // compiler-generated special constructor
  result += rhs;
  return result;
}

Complex operator*(double s, Complex const& rhs) {
  return Complex{s * rhs.real(), s * rhs.imag()};
}

c1 += c2
c1 + c2
2. * c1
```

Operator overloading (cont.)

```
class Complex {
  ...
  Complex& operator+=(Complex const& rhs) {
    r += rhs.r;
    i += rhs.i;
    return *this;
  }
};

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs; // compiler-generated special constructor
  result += rhs;
  return result;
}

Complex operator*(double s, Complex const& rhs) {
  return Complex{s * rhs.real(), s * rhs.imag()};
}

c1 += c2
c1 + c2
2. * c1
```

Operator overloading (cont.)

```
class Complex {
  ...
  Complex& operator+=(Complex const& rhs) {
    r += rhs.r;
    i += rhs.i;
    return *this;
  }
};

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs; // compiler-generated special constructor
  result += rhs;
  return result;
}

Complex operator*(double s, Complex const& rhs) {
  return Complex{s * rhs.real(), s * rhs.imag()};
}

c1 += c2
c1 + c2
2. * c1
```

Operator overloading (cont.)

```
class Complex {
  ...
  Complex& operator+=(Complex const& rhs) {
    r += rhs.r;
    i += rhs.i;
    return *this;
  }
};

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs; // compiler-generated special constructor
  result += rhs;
  return result;
}

Complex operator*(double s, Complex const& rhs) {
  return Complex{s * rhs.real(), s * rhs.imag()};
}

c1 += c2
c1 + c2
2. * c1
```

User-defined conversions

```
class Complex {
  Complex(double x = 0., double y = 0.) : r{x}, i{y} {}
  ...
};

double norm2(Complex const& c) { ... }

norm2(1.); // callable with a double (-> Complex)
norm2(1);  // callable with an int (-> double -> Complex)
3 + c;     // call operator+ with two Complex
```

- The one-argument constructor is used for the conversion
- An `explicit` constructor prevents the implicit conversion

```
class Complex {
  explicit Complex(double x = 0., double y = 0.) ...
  ...
}

norm2(1.);           // error
norm2(Complex{1.}); // ok
```

User-defined conversions

```
class Complex {
  Complex(double x = 0., double y = 0.) : r{x}, i{y} {}
  ...
};

double norm2(Complex const& c) { ... }

norm2(1.); // callable with a double (-> Complex)
norm2(1);  // callable with an int (-> double -> Complex)
3 + c;     // call operator+ with two Complex
```

- The one-argument constructor is used for the conversion
- An `explicit` constructor prevents the implicit conversion

```
class Complex {
  explicit Complex(double x = 0., double y = 0.) ...
  ...
}

norm2(1.);           // error
norm2(Complex{1.}); // ok
```

User-defined conversions

```
class Complex {
  Complex(double x = 0., double y = 0.) : r{x}, i{y} {}
  ...
};

double norm2(Complex const& c) { ... }

norm2(1.); // callable with a double (-> Complex)
norm2(1);  // callable with an int (-> double -> Complex)
3 + c;     // call operator+ with two Complex
```

- The one-argument constructor is used for the conversion
- An `explicit` constructor prevents the implicit conversion

```
class Complex {
  explicit Complex(double x = 0., double y = 0.) ...
  ...
}

norm2(1.);           // error
norm2(Complex{1.}); // ok
```

User-defined conversions

```
class Complex {
  Complex(double x = 0., double y = 0.) : r{x}, i{y} {}
  ...
};

double norm2(Complex const& c) { ... }

norm2(1.); // callable with a double (-> Complex)
norm2(1);  // callable with an int (-> double -> Complex)
3 + c;     // call operator+ with two Complex
```

- The one-argument constructor is used for the conversion
- An `explicit` constructor prevents the implicit conversion

```
class Complex {
  explicit Complex(double x = 0., double y = 0.) ...
  ...
}

norm2(1.);           // error
norm2(Complex{1.}); // ok
```

User-defined conversions

```
class Complex {
  Complex(double x = 0., double y = 0.) : r{x}, i{y} {}
  ...
};

double norm2(Complex const& c) { ... }

norm2(1.); // callable with a double (-> Complex)
norm2(1);  // callable with an int (-> double -> Complex)
3 + c;     // call operator+ with two Complex
```

- The one-argument constructor is used for the conversion
- An `explicit` constructor prevents the implicit conversion

```
class Complex {
  explicit Complex(double x = 0., double y = 0.) ...
  ...
}

norm2(1.);           // error
norm2(Complex{1.}); // ok
```

User-defined conversions

Conversions can go in the other direction too

```
struct Theta { /* similar to Rho */ };

struct Rho { // a wrapper around a double
  double value;
  explicit Rho(double v = 0.) : value{v} {}
  operator double() const { return value; }
};

class Complex {
  Complex(Rho rho, Theta theta)
      : r{rho.value * cos(theta.value)}
      , i{rho.value * sin(theta.value)}
  {}
  ...
};

Complex c{Rho{2.}, Theta{3.14 / 4}};
```

- The conversion operator can be explicit as well
- Avoid an implicit conversion in both constructor and conversion operator

User-defined conversions

Conversions can go in the other direction too

```
struct Theta { /* similar to Rho */ };

struct Rho { // a wrapper around a double
  double value;
  explicit Rho(double v = 0.) : value{v} {}
  operator double() const { return value; }
};

class Complex {
  Complex(Rho rho, Theta theta)
      : r{rho.value * cos(theta.value)}
      , i{rho.value * sin(theta.value)}
  {}
  ...
};

Complex c{Rho{2.}, Theta{3.14 / 4}};
```

- The conversion operator can be explicit as well
- Avoid an implicit conversion in both constructor and conversion operator

User-defined conversions

Conversions can go in the other direction too

```
struct Theta { /* similar to Rho */ };

struct Rho { // a wrapper around a double
  double value;
  explicit Rho(double v = 0.) : value{v} {}
  operator double() const { return value; }
};

class Complex {
  Complex(Rho rho, Theta theta)
      : r{rho * cos(theta)}
      , i{rho * sin(theta)}
  {}
  ...
};

Complex c{Rho{2.}, Theta{3.14 / 4}};
```

- The conversion operator can be explicit as well
- Avoid an implicit conversion in both constructor and conversion operator

User-defined conversions

Conversions can go in the other direction too

```
struct Theta { /* similar to Rho */ };

struct Rho { // a wrapper around a double
  double value;
  explicit Rho(double v = 0.) : value{v} {}
  operator double() const { return value; }
};

class Complex {
  Complex(Rho rho, Theta theta)
      : r{rho * cos(theta)}
      , i{rho * sin(theta)}
  {}
  ...
};

Complex c{Rho{2.}, Theta{3.14 / 4}};
```

- The conversion operator can be explicit as well
- Avoid an implicit conversion in both constructor and conversion operator

Namespaces

- Namespaces provide a mechanism to partition the space of names in a program in order to prevent conflicts
- The same namespace can be distributed among multiple blocks, possibly in different translation units
- Namespace can be nested

```
namespace mathlib {

  namespace util {
    // some utilities
  }

  class Complex { ... };
}

namespace mathlib { // possibly in another file

  double norm2(Complex const& c) { ... }

}
```

Namespaces

- Namespaces provide a mechanism to partition the space of names in a program in order to prevent conflicts
- The same namespace can be distributed among multiple blocks, possibly in different translation units
- Namespace can be nested

```
namespace mathlib {

  namespace util {
    // some utilities
  }

  class Complex { ... };
}

namespace mathlib { // possibly in another file

  double norm2(Complex const& c) { ... }

}
```

Namespaces

- Namespaces provide a mechanism to partition the space of names in a program in order to prevent conflicts
- The same namespace can be distributed among multiple blocks, possibly in different translation units
- Namespace can be nested

```
namespace mathlib {

  namespace util {
    // some utilities
  }

  class Complex { ... };
}

namespace mathlib { // possibly in another file

  double norm2(Complex const& c) { ... }

}
```

Name lookup

```
namespace mathlib {
  class Complex { ... };
}

Complex c;           // error
mathlib::Complex c; // ok

namespace ml = mathlib; // namespace alias
ml::Complex c;

using mathlib::Complex; // using declaration
Complex c;

using namespace mathlib; // using directive
Complex c;
```

`using` declarations and directives import one or all the names in a namespace in the current scope

- risk of name conflicts, use with care

Name lookup

```
namespace mathlib {
  class Complex { ... };
}

Complex c;           // error
mathlib::Complex c; // ok

namespace ml = mathlib; // namespace alias
ml::Complex c;

using mathlib::Complex; // using declaration
Complex c;

using namespace mathlib; // using directive
Complex c;
```

`using` declarations and directives import one or all the names in a namespace in the current scope

- risk of name conflicts, use with care

Name lookup

```
namespace mathlib {
  class Complex { ... };
}

Complex c;           // error
mathlib::Complex c; // ok

namespace ml = mathlib; // namespace alias
ml::Complex c;

using mathlib::Complex; // using declaration
Complex c;

using namespace mathlib; // using directive
Complex c;
```

`using` declarations and directives import one or all the names in a namespace in the current scope

- risk of name conflicts, use with care

Name lookup

```
namespace mathlib {
  class Complex { ... };
}

Complex c;           // error
mathlib::Complex c; // ok

namespace ml = mathlib; // namespace alias
ml::Complex c;

using mathlib::Complex; // using declaration
Complex c;

using namespace mathlib; // using directive
Complex c;
```

`using` declarations and directives import one or all the names in a namespace in the current scope

- risk of name conflicts, use with care

Name lookup

```
namespace mathlib {
  class Complex { ... };
}

Complex c;           // error
mathlib::Complex c; // ok

namespace ml = mathlib; // namespace alias
ml::Complex c;

using mathlib::Complex; // using declaration
Complex c;

using namespace mathlib; // using directive
Complex c;
```

`using` declarations and directives import one or all the names in a namespace in the current scope

- risk of name conflicts, use with care

Argument-dependent lookup

```
mathlib::Complex c1;
mathlib::Complex c2;
c1 + c2;
```

is equivalent to the function call

```
operator+(c1, c2);
```

- But `operator+` is actually `mathlib::operator+`. How can the compiler find it?
- To resolve an *unqualified* function name and build the *overload set*, the compiler searches also in the namespaces associated to the arguments of the call

Argument-dependent lookup

```
mathlib::Complex c1;
mathlib::Complex c2;
c1 + c2;
```

is equivalent to the function call

```
operator+(c1, c2);
```

- But `operator+` is actually `mathlib::operator+`. How can the compiler find it?
- To resolve an *unqualified* function name and build the *overload set*, the compiler searches also in the namespaces associated to the arguments of the call

Argument-dependent lookup

```
mathlib::Complex c1;
mathlib::Complex c2;
c1 + c2;
```

is equivalent to the function call

```
operator+(c1, c2);
```

- But `operator+` is actually `mathlib::operator+`. How can the compiler find it?
- To resolve an *unqualified* function name and build the *overload set*, the compiler searches also in the namespaces associated to the arguments of the call

Argument-dependent lookup

```
mathlib::Complex c1;
mathlib::Complex c2;
c1 + c2;
```

is equivalent to the function call

```
operator+(c1, c2);
```

- But `operator+` is actually `mathlib::operator+`. How can the compiler find it?
- To resolve an *unqualified* function name and build the *overload set*, the compiler searches also in the namespaces associated to the arguments of the call

Class template

What if we want a `Complex` with `float` members?

```
class Complex {
  double r;
  double i;
 public:
  Complex(
      double x = 0.
    , double y = 0.
  ) : r{x}, i{y} {}
  double real() const {return r;}
  double imag() const {return i;}
  Complex&
  operator+=(Complex const& o) {
    r += o.r;
    i += o.i;
    return *this;
  }
};
```

```
class Complex {
  float r;
  float i;
 public:
  Complex(
      float x = float{}
    , float y = float{}
  ) : r{x}, i{y} {}
  float real() const {return r;}
  float imag() const {return i;}
  Complex&
  operator+=(Complex const& o) {
    r += o.r;
    i += o.i;
    return *this;
  }
};
```

Class template

What if we want a `Complex` with `float` members?

```
class Complex {
  double r;
  double i;
 public:
  Complex(
      double x = 0.
    , double y = 0.
  ) : r{x}, i{y} {}
  double real() const {return r;}
  double imag() const {return i;}
  Complex&
  operator+=(Complex const& o) {
    r += o.r;
    i += o.i;
    return *this;
  }
};
```

```
class Complex {
  float r;
  float i;
 public:
  Complex(
      float x = 0.F
    , float y = 0.F
  ) : r{x}, i{y} {}
  float real() const {return r;}
  float imag() const {return i;}
  Complex&
  operator+=(Complex const& o) {
    r += o.r;
    i += o.i;
    return *this;
  }
};
```

Class template

What if we want a `Complex` with `float` members?

```
class Complex {
  double r;
  double i;
 public:
  Complex(
      double x = double{}
    , double y = double{}
  ) : r{x}, i{y} {}
  double real() const {return r;}
  double imag() const {return i;}
  Complex&
  operator+=(Complex const& o) {
    r += o.r;
    i += o.i;
    return *this;
  }
};
```

```
class Complex {
  float r;
  float i;
 public:
  Complex(
      float x = float{}
    , float y = float{}
  ) : r{x}, i{y} {}
  float real() const {return r;}
  float imag() const {return i;}
  Complex&
  operator+=(Complex const& o) {
    r += o.r;
    i += o.i;
    return *this;
  }
};
```

Class template

What if we want a `Complex` with `float` members?

```
class Complex {
  double r;
  double i;
 public:
  Complex(
      double x = double{}
    , double y = double{}
  ) : r{x}, i{y} {}
  double real() const {return r;}
  double imag() const {return i;}
  Complex&
  operator+=(Complex const& o) {
    r += o.r;
    i += o.i;
    return *this;
  }
};
```

```
class Complex {
  FP     r;
  FP     i;
 public:
  Complex(
      FP     x = FP    {}
    , FP     y = FP    {}
  ) : r{x}, i{y} {}
  FP     real() const {return r;}
  FP     imag() const {return i;}
  Complex&
  operator+=(Complex const& o) {
    r += o.r;
    i += o.i;
    return *this;
  }
};
```

Class template

What if we want a `Complex` with `float` members?

```
class Complex {
  double r;
  double i;
 public:
  Complex(
      double x = double{}
    , double y = double{}
  ) : r{x}, i{y} {}
  double real() const {return r;}
  double imag() const {return i;}
  Complex&
  operator+=(Complex const& o) {
    r += o.r;
    i += o.i;
    return *this;
  }
};
```

```
template<typename FP>
class Complex {
  FP    r;
  FP    i;
 public:
  Complex(
      FP    x = FP   {}
    , FP    y = FP   {}
  ) : r{x}, i{y} {}
  FP    real() const {return r;}
  FP    imag() const {return i;}
  Complex&
  operator+=(Complex const& o) {
    r += o.r;
    i += o.i;
    return *this;
  }
};
```

Class template (cont.)

```
template<typename FP>
class Complex {
  static_assert(std::is_floating_point<FP>::value);
  FP r;
  FP i;
 public:
  Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
  FP   real() const { return r; } // deduce the return type
  FP   imag() const { return i; }
  Complex& operator+=(Complex const& o)
  { ... }
};

Complex c;                      // error
Complex<float> cf;              // type is Complex<float>
Complex<double> cd;             // different type than cf
cd += Complex<double>{1., 2.}; // ok
cd += Complex<float>{1., 2.};  // error
Complex<int> ci;                // error
```

Class template (cont.)

```
template<typename FP>
class Complex {
  static_assert(std::is_floating_point<FP>::value);
  FP r;
  FP i;
 public:
  Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
  auto real() const { return r; } // deduce the return type
  auto imag() const { return i; }
  Complex& operator+=(Complex const& o)
  { ... }
};

Complex c;                      // error
Complex<float> cf;              // type is Complex<float>
Complex<double> cd;             // different type than cf
cd += Complex<double>{1., 2.}; // ok
cd += Complex<float>{1., 2.};  // error
Complex<int> ci;                // error
```

Class template (cont.)

```
template<typename FP>
class Complex {
  static_assert(std::is_floating_point<FP>::value);
  FP r;
  FP i;
 public:
  Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
  auto real() const { return r; } // deduce the return type
  auto imag() const { return i; }
  Complex& operator+=(Complex const& o)
  { ... }
};

Complex c;                          // error
Complex<float> cf;                  // type is Complex<float>
Complex<double> cd;                 // different type than cf
cd += Complex<double>{1., 2.}; // ok
cd += Complex<float>{1., 2.};  // error
Complex<int> ci;                    // error
```

Class template (cont.)

```
template<typename FP>
class Complex {
  static_assert(std::is_floating_point<FP>::value);
  FP r;
  FP i;
 public:
  Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
  auto real() const { return r; } // deduce the return type
  auto imag() const { return i; }
  Complex& operator+=(Complex const& o)
  { ... }
};

Complex c;                          // error
Complex<float> cf;                  // type is Complex<float>
Complex<double> cd;                 // different type than cf
cd += Complex<double>{1., 2.}; // ok
cd += Complex<float>{1., 2.};  // error
Complex<int> ci;                    // error
```

Class template (cont.)

```
template<typename FP>
class Complex {
  static_assert(std::is_floating_point<FP>::value);
  FP r;
  FP i;
 public:
  Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
  auto real() const { return r; } // deduce the return type
  auto imag() const { return i; }
  Complex& operator+=(Complex const& o)
  { ... }
};

Complex c;                          // error
Complex<float> cf;                  // type is Complex<float>
Complex<double> cd;                 // different type than cf
cd += Complex<double>{1., 2.}; // ok
cd += Complex<float>{1., 2.};  // error
Complex<int> ci;                    // error
```

Class template (cont.)

```
template<typename FP>
class Complex {
  static_assert(std::is_floating_point<FP>::value);
  FP r;
  FP i;
 public:
  Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
  auto real() const { return r; } // deduce the return type
  auto imag() const { return i; }
  Complex& operator+=(Complex const& o)
  { ... }
};

Complex c;                          // error
Complex<float> cf;                  // type is Complex<float>
Complex<double> cd;                 // different type than cf
cd += Complex<double>{1., 2.}; // ok
cd += Complex<float>{1., 2.};  // error
Complex<int> ci;                    // error
```

Class template (cont.)

```
template<typename FP>
class Complex {
  static_assert(std::is_floating_point<FP>::value);
  FP r;
  FP i;
 public:
  Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
  auto real() const { return r; } // deduce the return type
  auto imag() const { return i; }
  Complex& operator+=(Complex const& o)
  { ... }
};

Complex c;                          // error
Complex<float> cf;                  // type is Complex<float>
Complex<double> cd;                 // different type than cf
cd += Complex<double>{1., 2.}; // ok
cd += Complex<float>{1., 2.};  // error
Complex<int> ci;                    // acceptable?
```

Class template (cont.)

```
template<typename FP>
class Complex {
  static_assert(std::is_floating_point<FP>::value);
  FP r;
  FP i;
 public:
  Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
  auto real() const { return r; } // deduce the return type
  auto imag() const { return i; }
  Complex& operator+=(Complex const& o)
  { ... }
};

Complex c;                          // error
Complex<float> cf;                  // type is Complex<float>
Complex<double> cd;                 // different type than cf
cd += Complex<double>{1., 2.}; // ok
cd += Complex<float>{1., 2.};  // error
Complex<int> ci;                    // error
```

Function template

```
double norm2(Complex const& c) { ... }
norm2(cf); // error; cf is of type Complex<float>
```

```
auto norm2(Complex<float> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cf); // ok
norm2(cd); // error; cd is of type Complex<double>
```

```
auto norm2(Complex<double> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cd); // ok
```

```
template<typename FP>
auto norm2(Complex<FP> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
auto nf = norm2(cf); // nf is of type float
auto nd = norm2(cd); // nd is of type double
```

Function template

```
double norm2(Complex const& c) { ... }
norm2(cf); // error; cf is of type Complex<float>
```

```
auto norm2(Complex<float> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cf); // ok
norm2(cd); // error; cd is of type Complex<double>
```

```
auto norm2(Complex<double> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cd); // ok
```

```
template<typename FP>
auto norm2(Complex<FP> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
auto nf = norm2(cf); // nf is of type float
auto nd = norm2(cd); // nd is of type double
```

Function template

```
double norm2(Complex const& c) { ... }
norm2(cf); // error; cf is of type Complex<float>
```

```
auto norm2(Complex<float> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cf); // ok
norm2(cd); // error; cd is of type Complex<double>
```

```
auto norm2(Complex<double> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cd); // ok
```

```
template<typename FP>
auto norm2(Complex<FP> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
auto nf = norm2(cf); // nf is of type float
auto nd = norm2(cd); // nd is of type double
```

Function template

```
double norm2(Complex const& c) { ... }
norm2(cf); // error; cf is of type Complex<float>
```

```
auto norm2(Complex<float> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cf); // ok
norm2(cd); // error; cd is of type Complex<double>
```

```
auto norm2(Complex<double> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cd); // ok
```

```
template<typename FP>
auto norm2(Complex<FP> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
auto nf = norm2(cf); // nf is of type float
auto nd = norm2(cd); // nd is of type double
```

Function template

```
double norm2(Complex const& c) { ... }
norm2(cf); // error; cf is of type Complex<float>
```

```
auto norm2(Complex<float> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cf); // ok
norm2(cd); // error; cd is of type Complex<double>
```

```
auto norm2(Complex<double> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cd); // ok
```

```
template<typename FP>
auto norm2(Complex<FP> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
auto nf = norm2(cf); // nf is of type float
auto nd = norm2(cd); // nd is of type double
```

Function template

```
double norm2(Complex const& c) { ... }
norm2(cf); // error; cf is of type Complex<float>
```

```
auto norm2(Complex<float> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cf); // ok
norm2(cd); // error; cd is of type Complex<double>
```

```
auto norm2(Complex<double> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
norm2(cd); // ok
```

```
template<typename FP>
auto norm2(Complex<FP> const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}
auto nf = norm2(cf); // nf is of type float
auto nd = norm2(cd); // nd is of type double
```

Function template (cont.)

```
template<typename FP>
auto norm2(Complex<FP> const& c) { ... }

template<typename C>
auto norm2(C const& c)
{ return c.real() * c.real() + c.imag() * c.imag(); }

norm2(cf); // ok
norm2(cd); // ok
std::complex<float> scf;
norm2(scf); // ok!

namespace std {
  template<class T>
  class complex {
    ...
   public:
    T real() const;
    T imag() const;
  };
}
```

Function template (cont.)

```
template<typename FP>
auto norm2(Complex<FP> const& c) { ... }

template<typename C>
auto norm2(C const& c)
{ return c.real() * c.real() + c.imag() * c.imag(); }

norm2(cf); // ok
norm2(cd); // ok
std::complex<float> scf;
norm2(scf); // ok!

namespace std {
  template<class T>
  class complex {
    ...
   public:
    T real() const;
    T imag() const;
  };
}
```

Function template (cont.)

```
template<typename FP>
auto norm2(Complex<FP> const& c) { ... }

template<typename C>
auto norm2(C const& c)
{ return c.real() * c.real() + c.imag() * c.imag(); }

norm2(cf); // ok
norm2(cd); // ok
std::complex<float> scf;
norm2(scf); // ok!

namespace std {
  template<class T>
  class complex {
    ...
   public:
    T real() const;
    T imag() const;
  };
}
```

Function template (cont.)

```
template<typename FP>
auto norm2(Complex<FP> const& c) { ... }

template<typename C>
auto norm2(C const& c)
{ return c.real() * c.real() + c.imag() * c.imag(); }

norm2(cf); // ok
norm2(cd); // ok
std::complex<float> scf;
norm2(scf); // ok!

namespace std {
  template<class T>
  class complex {
    ...
   public:
    T real() const;
    T imag() const;
  };
}
```

Function template (cont.)

```
template<typename FP>
auto norm2(Complex<FP> const& c) { ... }

template<typename C>
auto norm2(C const& c)
{ return c.real() * c.real() + c.imag() * c.imag(); }

norm2(cf); // ok
norm2(cd); // ok
std::complex<float> scf;
norm2(scf); // ok!

namespace std {
  template<class T>
  class complex {
    ...
   public:
    T real() const;
    T imag() const;
  };
}
```

Function template (cont.)

```
template<typename FP>
auto norm2(Complex<FP> const& c) { ... }

template<typename C>
auto norm2(C const& c)
{ return c.real() * c.real() + c.imag() * c.imag(); }

norm2(cf); // ok
norm2(cd); // ok
std::complex<float> scf;
norm2(scf); // ok!

namespace std {
  template<class T>
  class complex {
    ...
   public:
    T real() const;
    T imag() const;
  };
}
```

Function template (cont.)

```
template<typename FP>
auto norm2(Complex<FP> const& c) { ... }

template<typename C>
auto norm2(C const& c)
{ return c.real() * c.real() + c.imag() * c.imag(); }

norm2(cf); // ok
norm2(cd); // ok
std::complex<float> scf;
norm2(scf); // ok!

namespace std {
  template<class T>
  class complex {
    ...
   public:
    T real() const;
    T imag() const;
  };
}
```

This shows the basic idea behind generic programming

Generic programming

- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy

```
namespace std {
  template<class T>
  T const&
  min(T const& a, T const& b)
  { return a < b ? a : b; }
}

auto i = 0;
auto j = 1;
auto& k = std::min(i, j); // ok
assert(&k == &i);

auto d = 1.;
auto e = 0.;
auto& f = std::min(d, e); // ok
assert(&f == &e);
```

```
std::complex<float> r{0., 0.};
std::complex<float> s{1., 0.};
std::min(r, s); // error

struct T {};
bool
operator<(T const&, T const&) {
  return false;
}

T v, w;
auto& z = std::min(v, w); // ok
assert(&z == &v);
```

Generic programming

- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy

```
namespace std {
  template<class T>
  T const&
  min(T const& a, T const& b)
  { return a < b ? a : b; }
}

auto i = 0;
auto j = 1;
auto& k = std::min(i, j); // ok
assert(&k == &i);

auto d = 1.;
auto e = 0.;
auto& f = std::min(d, e); // ok
assert(&f == &e);
```

```
std::complex<float> r{0., 0.};
std::complex<float> s{1., 0.};
std::min(r, s); // error

struct T {};
bool
operator<(T const&, T const&) {
  return false;
}

T v, w;
auto& z = std::min(v, w); // ok
assert(&z == &v);
```

Generic programming

- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy

```
namespace std {
  template<class T>
  T const&
  min(T const& a, T const& b)
  { return a < b ? a : b; }
}

auto i = 0;
auto j = 1;
auto& k = std::min(i, j); // ok
assert(&k == &i);

auto d = 1.;
auto e = 0.;
auto& f = std::min(d, e); // ok
assert(&f == &e);
```

```
std::complex<float> r{0., 0.};
std::complex<float> s{1., 0.};
std::min(r, s); // error

struct T {};
bool
operator<(T const&, T const&) {
  return false;
}

T v, w;
auto& z = std::min(v, w); // ok
assert(&z == &v);
```

Generic programming

- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy

```
namespace std {
  template<class T>
  T const&
  min(T const& a, T const& b)
  { return a < b ? a : b; }
}

auto i = 0;
auto j = 1;
auto& k = std::min(i, j); // ok
assert(&k == &i);

auto d = 1.;
auto e = 0.;
auto& f = std::min(d, e); // ok
assert(&f == &e);
```

```
std::complex<float> r{0., 0.};
std::complex<float> s{1., 0.};
std::min(r, s); // error

struct T {};
bool
operator<(T const&, T const&) {
  return false;
}

T v, w;
auto& z = std::min(v, w); // ok
assert(&z == &v);
```

Generic programming

- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy

```
namespace std {
  template<class T>
  T const&
  min(T const& a, T const& b)
  { return a < b ? a : b; }
}

auto i = 0;
auto j = 1;
auto& k = std::min(i, j); // ok
assert(&k == &i);

auto d = 1.;
auto e = 0.;
auto& f = std::min(d, e); // ok
assert(&f == &e);
```

```
std::complex<float> r{0., 0.};
std::complex<float> s{1., 0.};
std::min(r, s); // error

struct T {};
bool
operator<(T const&, T const&) {
  return false;
}

T v, w;
auto& z = std::min(v, w); // ok
assert(&z == &v);
```

Generic programming

- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy

```
namespace std {
  template<class T>
  T const&
  min(T const& a, T const& b)
  { return a < b ? a : b; }
}

auto i = 0;
auto j = 1;
auto& k = std::min(i, j); // ok
assert(&k == &i);

auto d = 1.;
auto e = 0.;
auto& f = std::min(d, e); // ok
assert(&f == &e);
```

```
std::complex<float> r{0., 0.};
std::complex<float> s{1., 0.};
std::min(r, s); // error

struct T {};
bool
operator<(T const&, T const&) {
  return false;
}

T v, w;
auto& z = std::min(v, w); // ok
assert(&z == &v);
```

Generic programming

- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy

```
namespace std {
  template<class T>
  T const&
  min(T const& a, T const& b)
  { return a < b ? a : b; }
}

auto i = 0;
auto j = 1;
auto& k = std::min(i, j); // ok
assert(&k == &i);

auto d = 1.;
auto e = 0.;
auto& f = std::min(d, e); // ok
assert(&f == &e);
```

```
std::complex<float> r{0., 0.};
std::complex<float> s{1., 0.};
std::min(r, s); // error

struct T {};
bool
operator<(T const&, T const&) {
  return false;
}

T v, w;
auto& z = std::min(v, w); // ok
assert(&z == &v);
```

Generic programming

- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy

```
namespace std {
  template<class T>
  T const&
  min(T const& a, T const& b)
  { return a < b ? a : b; }
}

auto i = 0;
auto j = 1;
auto& k = std::min(i, j); // ok
assert(&k == &i);

auto d = 1.;
auto e = 0.;
auto& f = std::min(d, e); // ok
assert(&f == &e);
```

```
std::complex<float> r{0., 0.};
std::complex<float> s{1., 0.};
std::min(r, s); // error

struct T {};
bool
operator<(T const&, T const&) {
  return false;
}

T v, w;
auto& z = std::min(v, w); // ok
assert(&z == &v);
```

Type deduction

Note that we have to specify the template argument when instantiating a class template, but not a function template

```
template<class C> class Complex { ... };
template<class C> auto norm2(C const& c) { ... }

Complex<double> d{1., 2.};
Complex e{1., 2.};          // error
norm2(d);                   // ok
norm2<double>(d);           // ok, but not needed
norm2<float>(d);            // force the float instantiation
```

Let's use a factory function template

```
template<class FP> auto make_complex(FP x, FP y)
{ return Complex<FP>{x, y}; }

auto e = make_complex(1., 2.);   // Complex<double>
auto f = make_complex(1.F, 2.F); // Complex<float>
auto g = make_complex(1., 2.F);  // error
```

In the standard library there are many `make_something` function templates

Type deduction

Note that we have to specify the template argument when instantiating a class template, but not a function template

```
template<class C> class Complex { ... };
template<class C> auto norm2(C const& c) { ... }

Complex<double> d{1., 2.};
Complex e{1., 2.};          // error
norm2(d);                   // ok
norm2<double>(d);           // ok, but not needed
norm2<float>(d);            // force the float instantiation
```

Let's use a factory function template

```
template<class FP> auto make_complex(FP x, FP y)
{ return Complex<FP>{x, y}; }

auto e = make_complex(1., 2.);   // Complex<double>
auto f = make_complex(1.F, 2.F); // Complex<float>
auto g = make_complex(1., 2.F);  // error
```

In the standard library there are many `make_something` function templates

Type deduction

Note that we have to specify the template argument when instantiating a class template, but not a function template

```
template<class C> class Complex { ... };
template<class C> auto norm2(C const& c) { ... }

Complex<double> d{1., 2.};
Complex e{1., 2.};          // error
norm2(d);                   // ok
norm2<double>(d);           // ok, but not needed
norm2<float>(d);            // force the float instantiation
```

Let's use a factory function template

```
template<class FP> auto make_complex(FP x, FP y)
{ return Complex<FP>{x, y}; }

auto e = make_complex(1., 2.);   // Complex<double>
auto f = make_complex(1.F, 2.F); // Complex<float>
auto g = make_complex(1., 2.F);  // error
```

In the standard library there are many `make_something` function templates

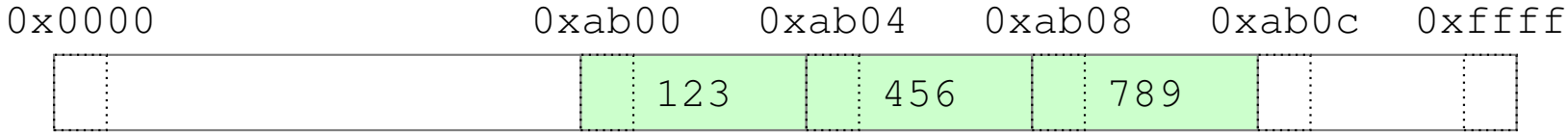
Arrays

Contiguous sequence of homogeneous objects in memory

```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

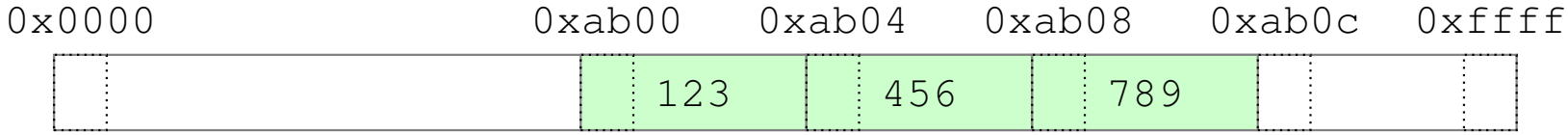
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

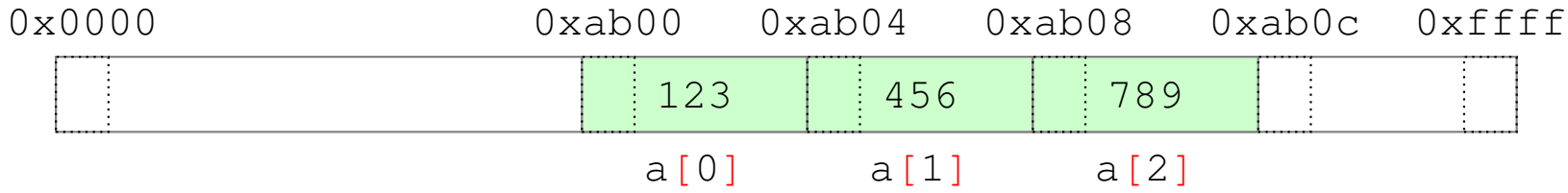
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

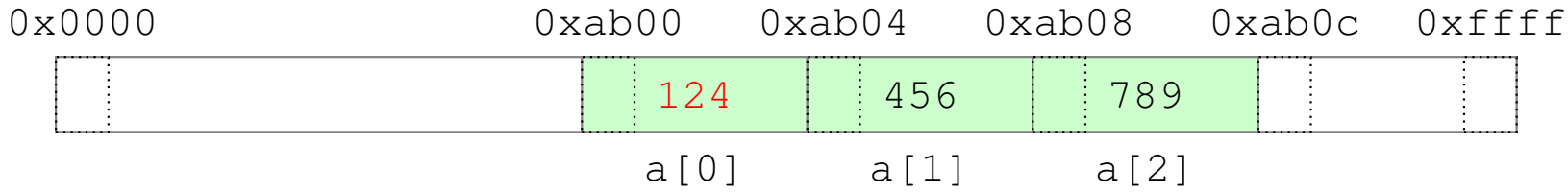
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

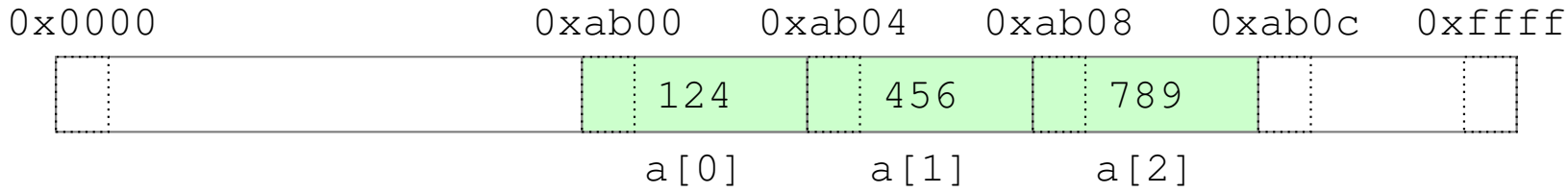
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

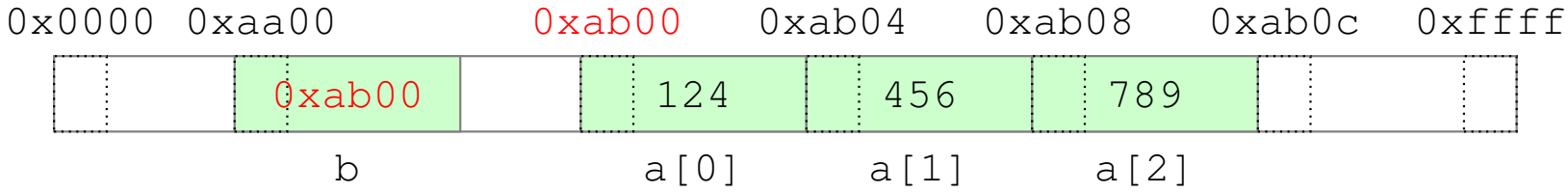
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

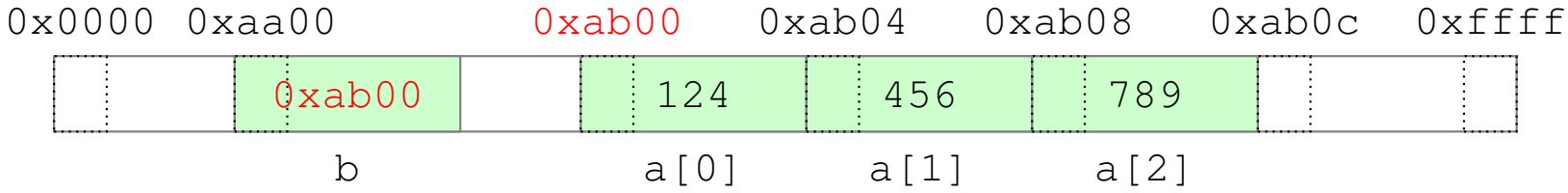
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

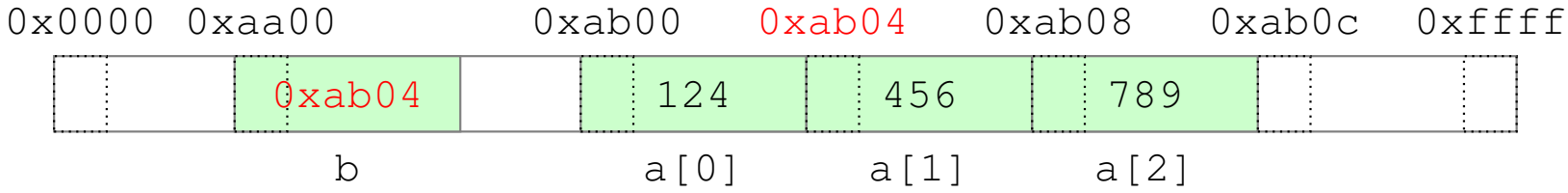
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

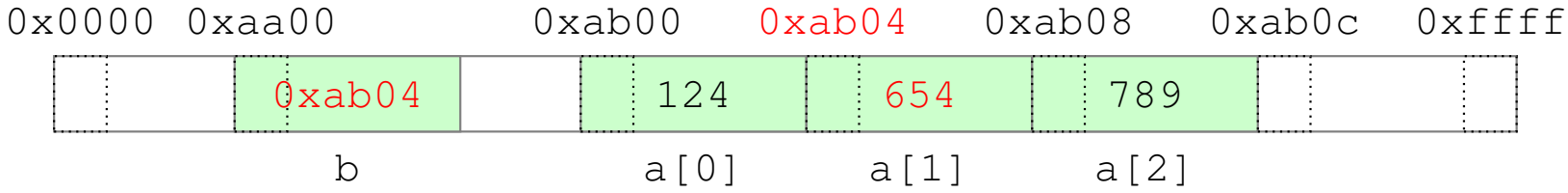
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

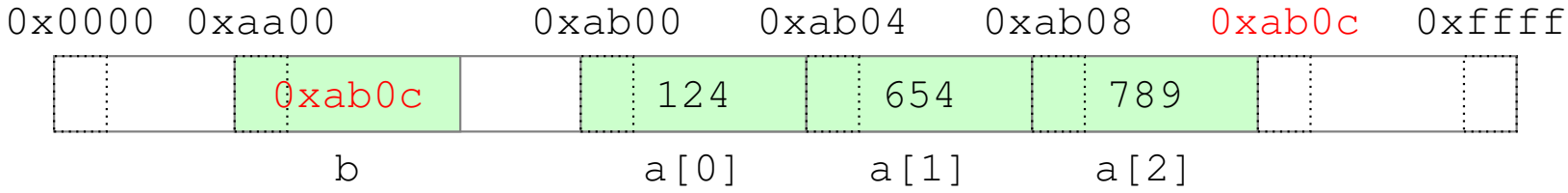
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

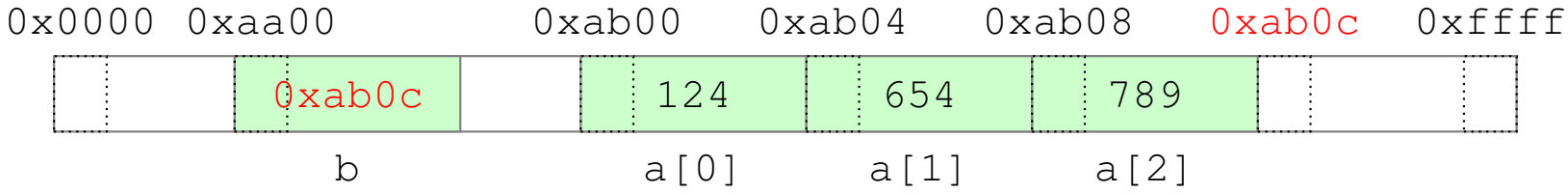
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

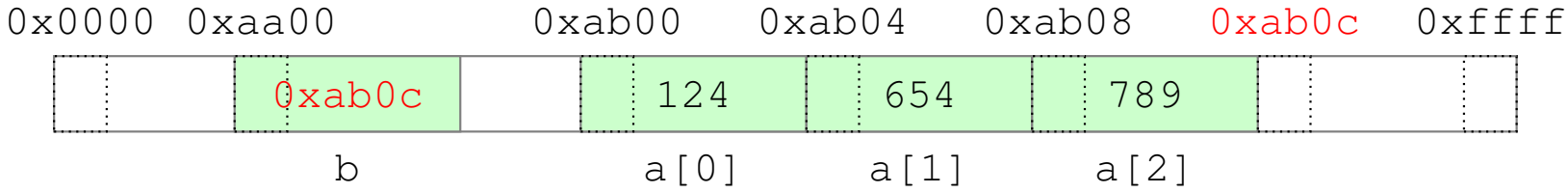
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Arrays

Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

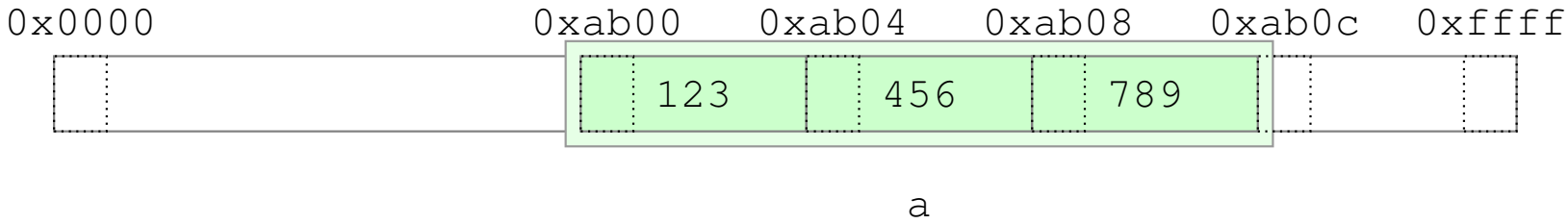
std::array

Type-safe, no-runtime overhead alternative to a native array

```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                          // undefined behavior
auto b = a;                    // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                           // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                        // increase by 2 * sizeof(int)
*p;                            // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

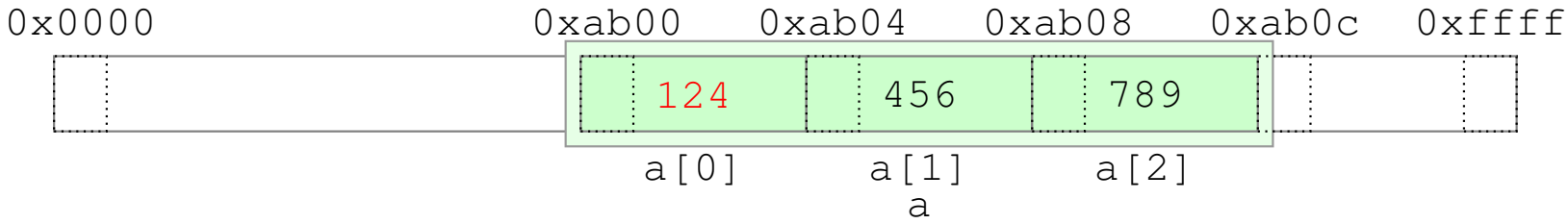
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

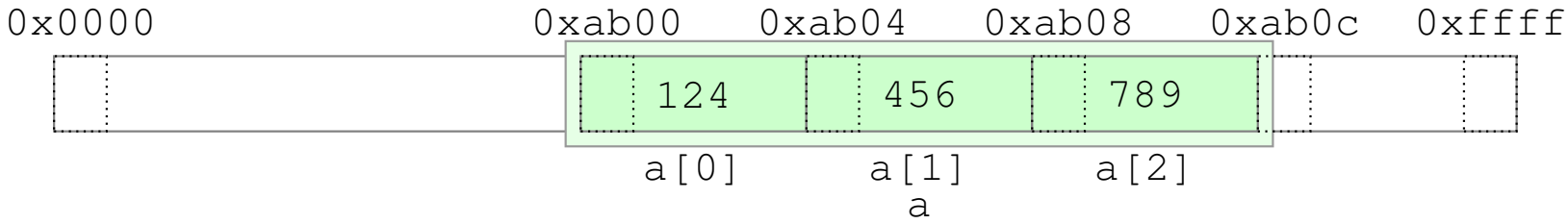
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

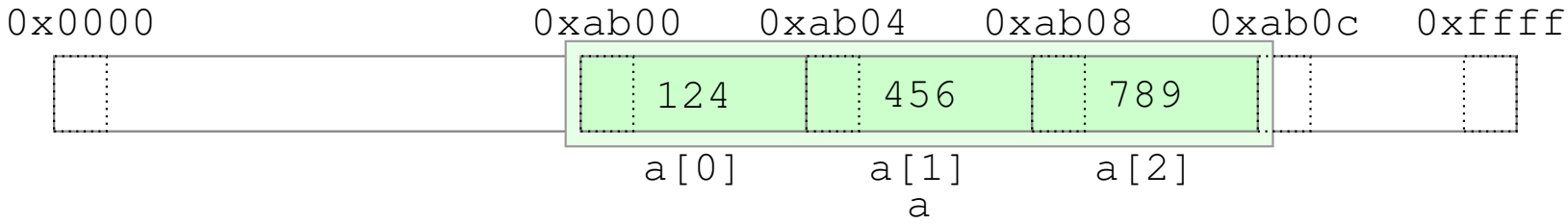
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

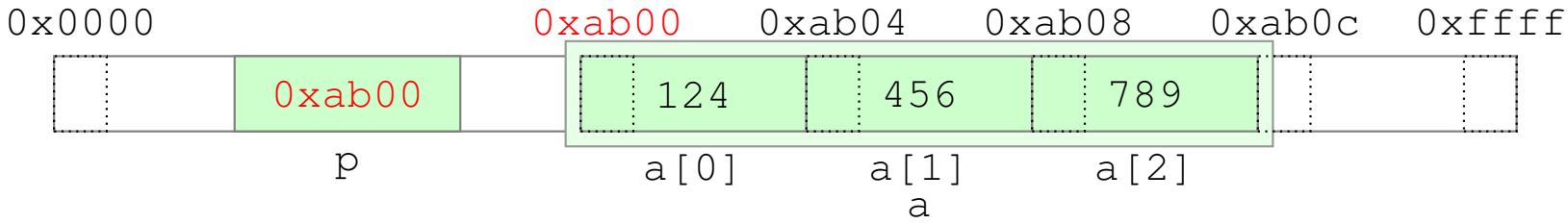
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

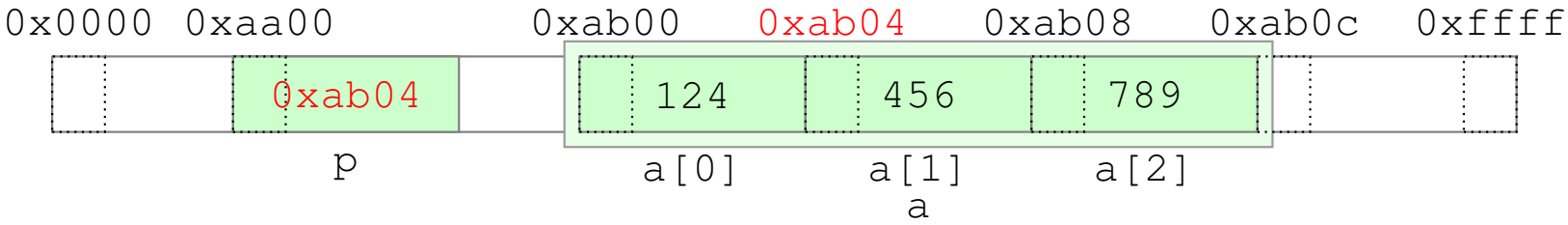
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

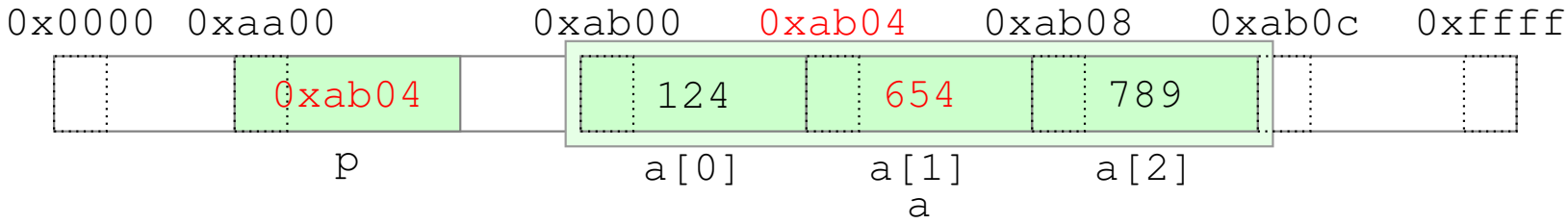
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

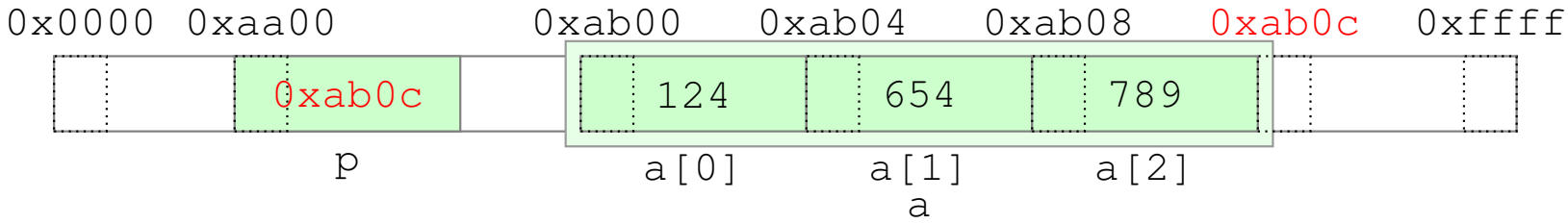
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

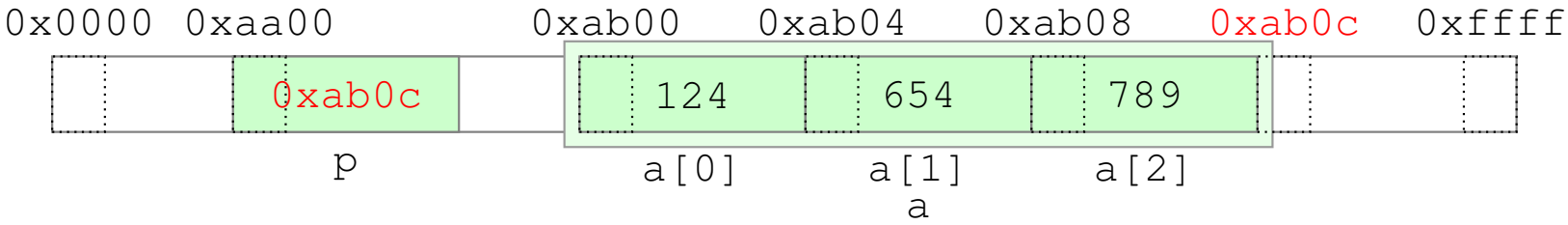
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

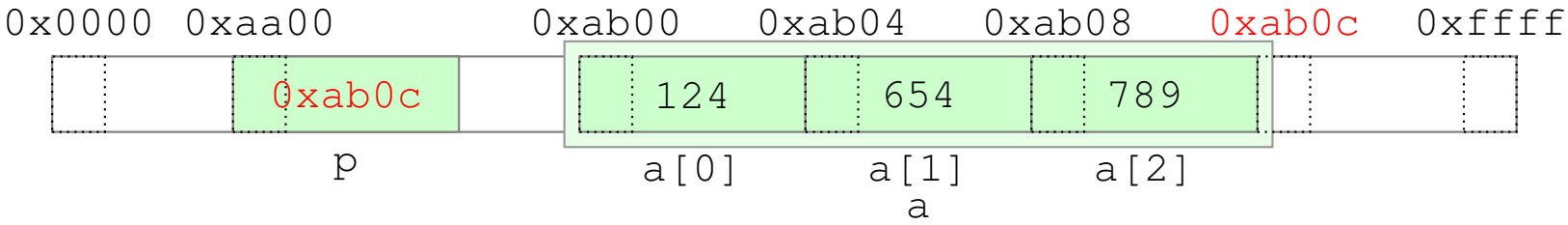
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

```
template<class T, unsigned N> class array {
  T elems[N]; // no space overhead wrt to native array
public:
  size_t size() const { return N; }
  T* data() { return elems; }
  T const* data() const { return elems; }
  T& operator[](unsigned n) { return elems[n]; } // no checks!
  T const& operator[](unsigned n) const { return elems[n]; }
  ...
};

template<class T, unsigned N>
bool operator==(array<T, N> const& l, array<T, N> const& r) {
  return std::equal(l.data(), l.data() + N, r.data());
}

template<class T, unsigned N>
bool operator<(array<T, N> const& l, array<T, N> const& r) {
  return std::lexicographical_compare(
      l.data(), l.data() + N
    , r.data(), r.data() + N
  );
}
...
```

The C++ standard library

- The standard library contains components of general use
  - containers (data structures)
  - algorithms
  - strings
  - input/output
  - random numbers
  - regular expressions
  - concurrency and parallelism
  - filesystem
  - ...
- The subset containing containers and algorithms is known as STL (Standard Template Library)
- But templates are everywhere

The C++ standard library

- The standard library contains components of general use
  - containers (data structures)
  - algorithms
  - strings
  - input/output
  - random numbers
  - regular expressions
  - concurrency and parallelism
  - filesystem
  - ...
- The subset containing containers and algorithms is known as STL (Standard Template Library)
- But templates are everywhere

The C++ standard library

- The standard library contains components of general use
  - containers (data structures)
  - algorithms
  - strings
  - input/output
  - random numbers
  - regular expressions
  - concurrency and parallelism
  - filesystem
  - ...
- The subset containing containers and algorithms is known as STL (Standard Template Library)
- But templates are everywhere

Containers

- Objects that contain and own other objects
- Different characteristics and operations, some common traits
- Implemented as class templates

Sequence The client decides where an element gets inserted

- `array`, `vector`, `list`, ...

Associative The container decides where an element gets inserted

Ordered The elements are sorted

- `map`, `multimap`, `set`, `multiset`

Unordered The elements are hashed

- `unordered_*`

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying all_of any_of for_each count
count_if mismatch equal find find_if
adjacent_find search ...

Modifying copy fill generate transform remove
replace swap reverse rotate shuffle
sample unique ...

Partitioning partition stable_partition ...

Sorting sort partial_sort nth_element ...

Set set_union set_intersection
set_difference ...

Min/Max min max minmax
lexicographical_compare clamp ...

Numeric iota accumulate inner_product
partial_sum adjacent_difference ...

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`
Min/Max	`min max minmax lexicographical_compare clamp ...`
Numeric	`iota accumulate inner_product partial_sum adjacent_difference ...`

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`
Min/Max	`min max minmax lexicographical_compare clamp ...`
Numeric	`iota accumulate inner_product partial_sum adjacent_difference ...`

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`
Min/Max	`min max minmax lexicographical_compare clamp ...`
Numeric	`iota accumulate inner_product partial_sum adjacent_difference ...`

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`
Min/Max	`min max minmax lexicographical_compare clamp ...`
Numeric	`iota accumulate inner_product partial_sum adjacent_difference ...`

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`
Min/Max	`min max minmax lexicographical_compare clamp ...`
Numeric	`iota accumulate inner_product partial_sum adjacent_difference ...`

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`
Min/Max	`min max minmax lexicographical_compare clamp ...`
Numeric	`iota accumulate inner_product partial_sum adjacent_difference ...`

Algorithms

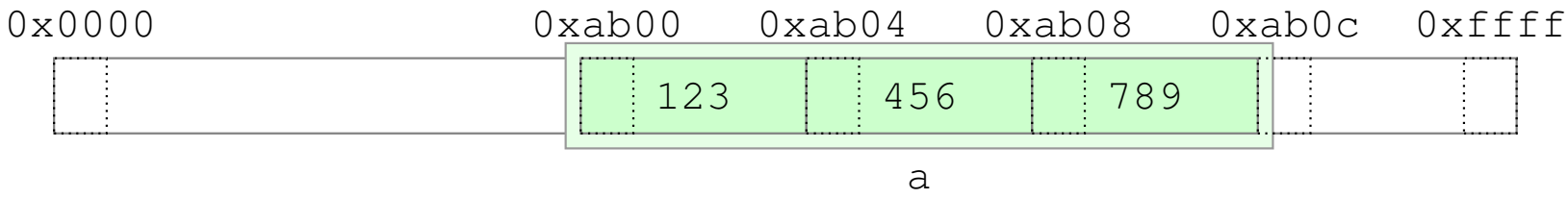
- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`
Min/Max	`min max minmax lexicographical_compare clamp ...`
Numeric	`iota accumulate inner_product partial_sum adjacent_difference ...`

Range

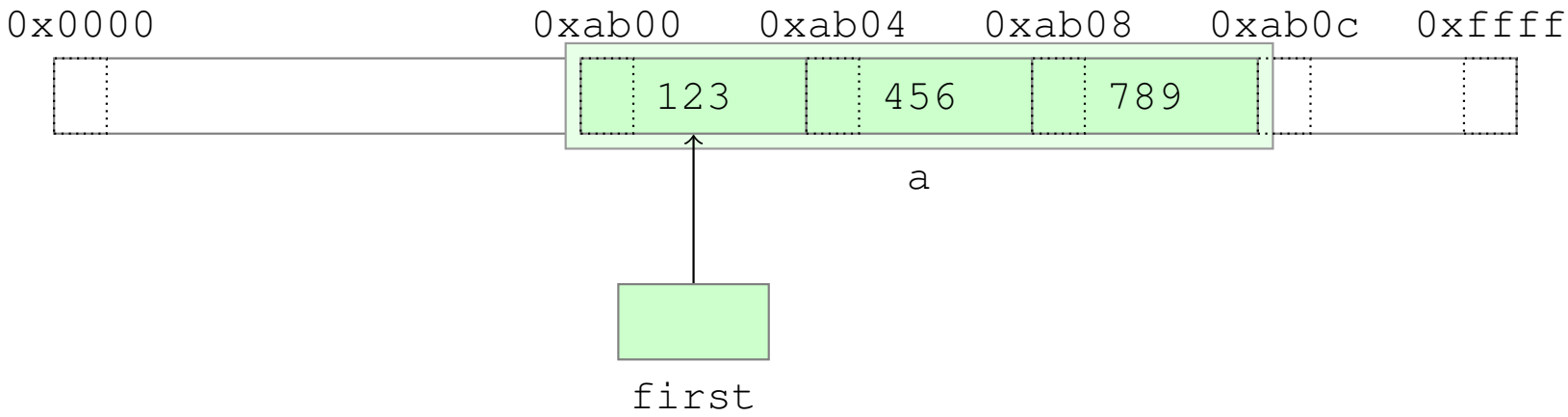
- A range is defined by a pair of iterators [*first*, *last*), with *last* referring to one past the last element in the range
  - the range is *half-open*
  - *first* == *last* means the range is empty
  - *last* can be used to return failure
- An iterator is a generalization of a pointer
  - it supports the same operations, possibly through overloaded operators
  - certainly `*` `++` `->`, maybe `-` `+=` `-=`
- Ranges are typically obtained from containers calling specific methods

Range (cont.)



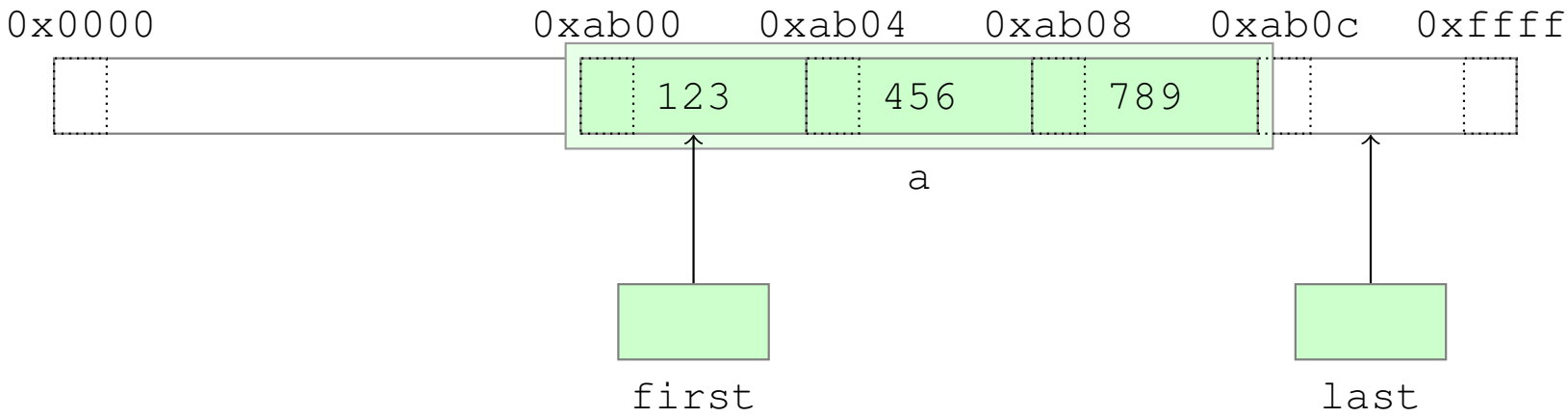

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



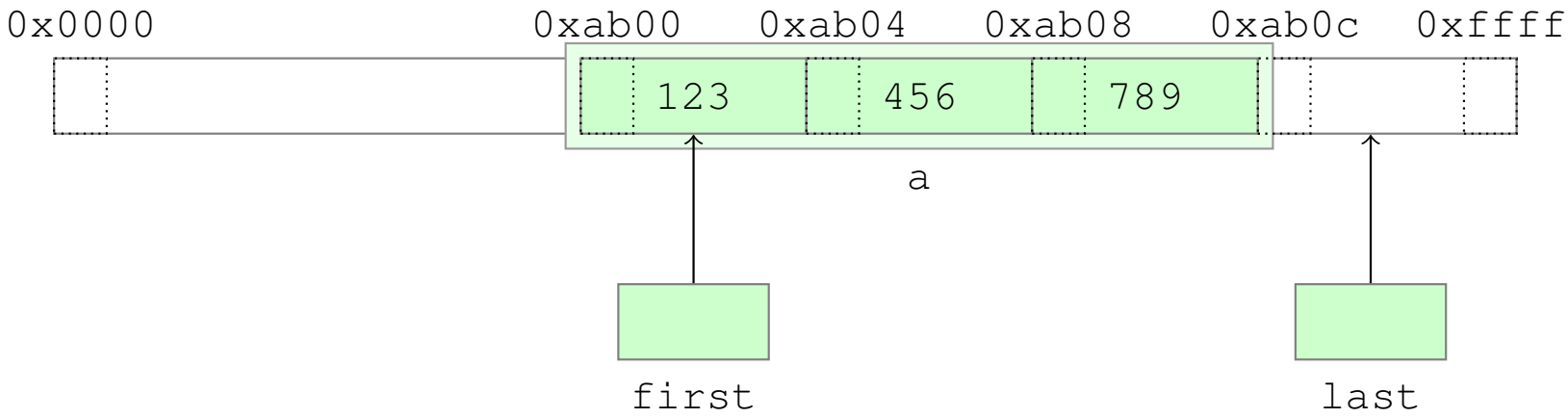

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



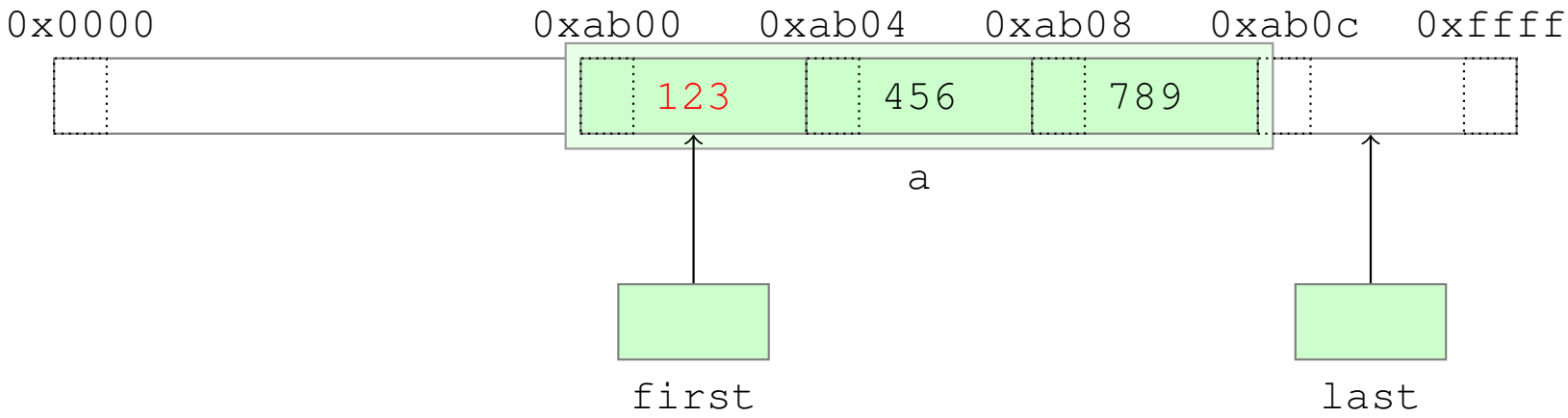

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



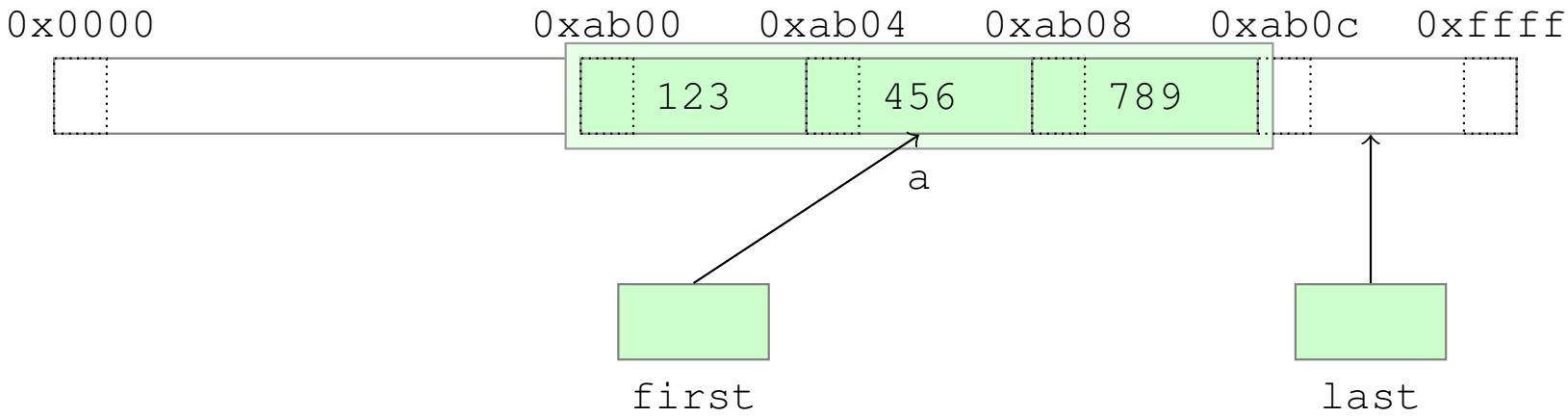

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



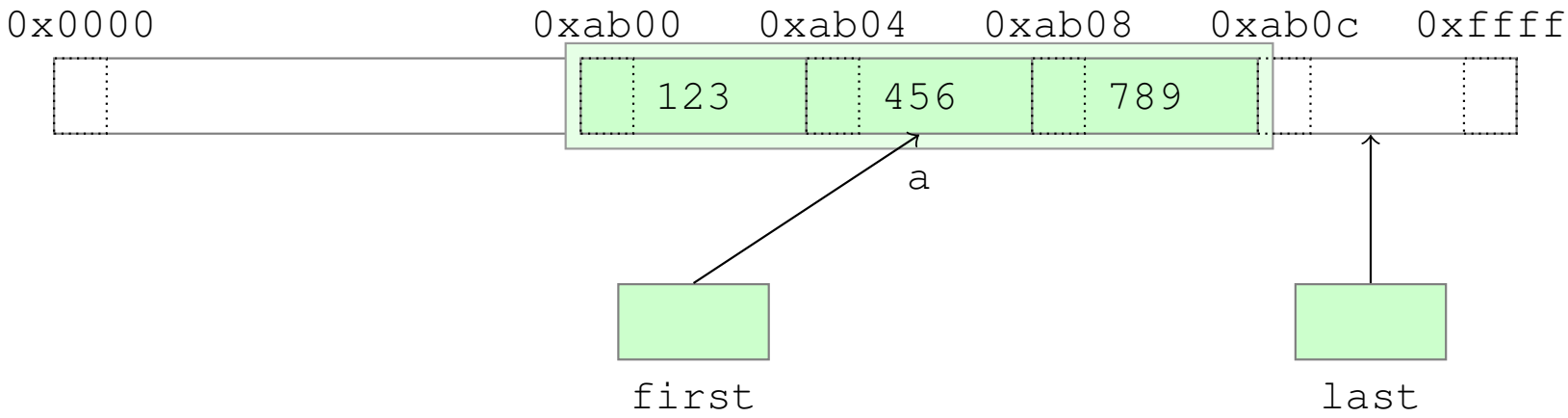

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



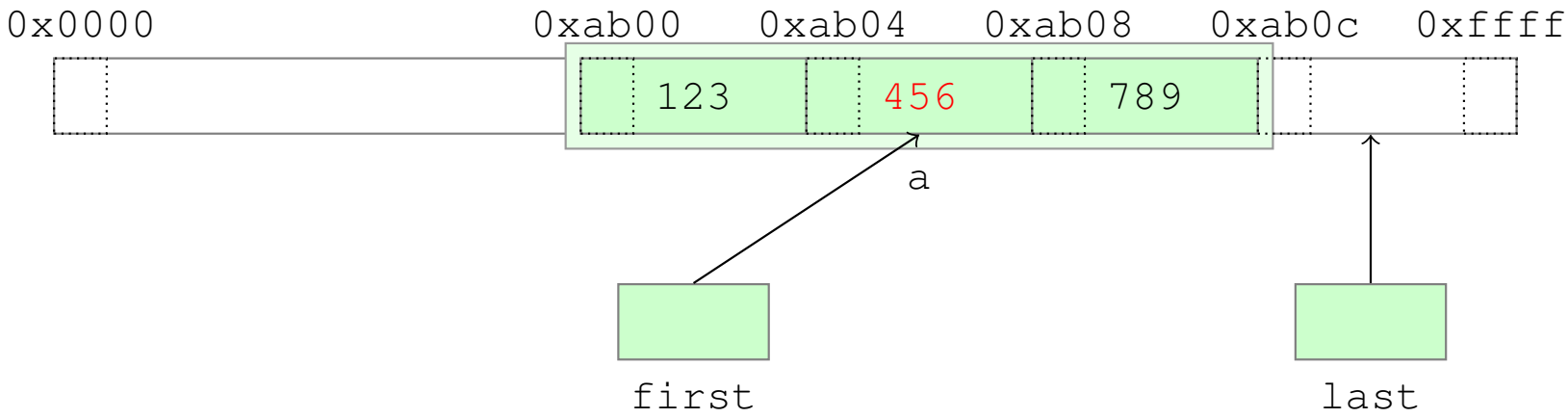

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



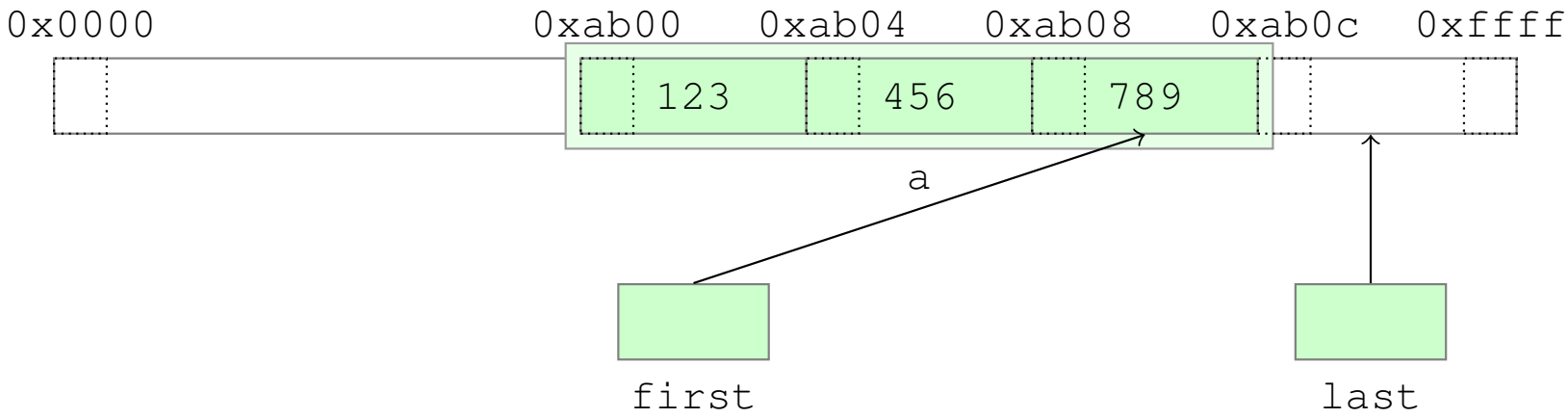

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



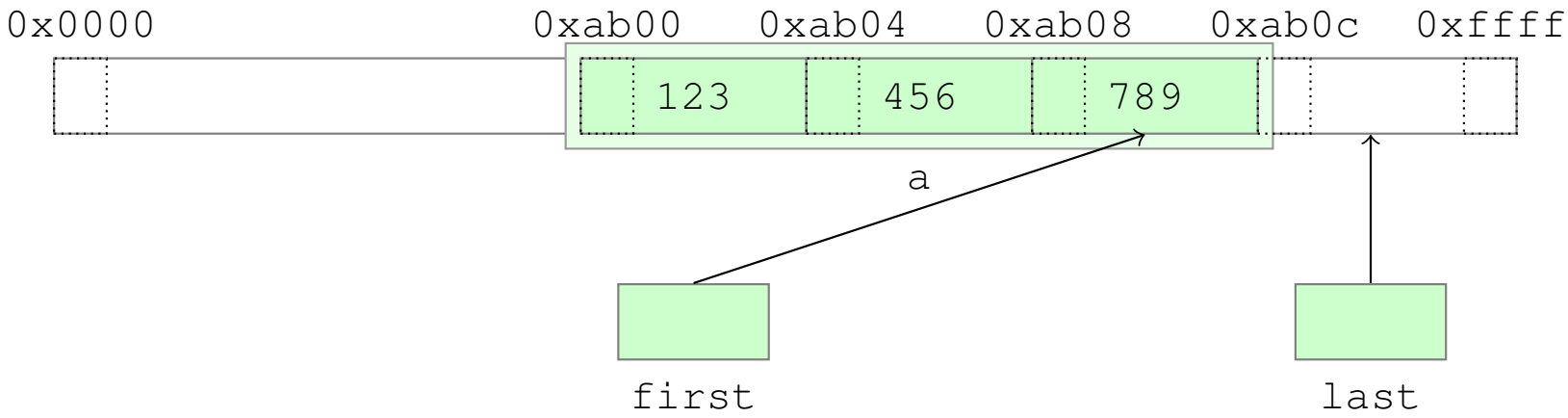

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



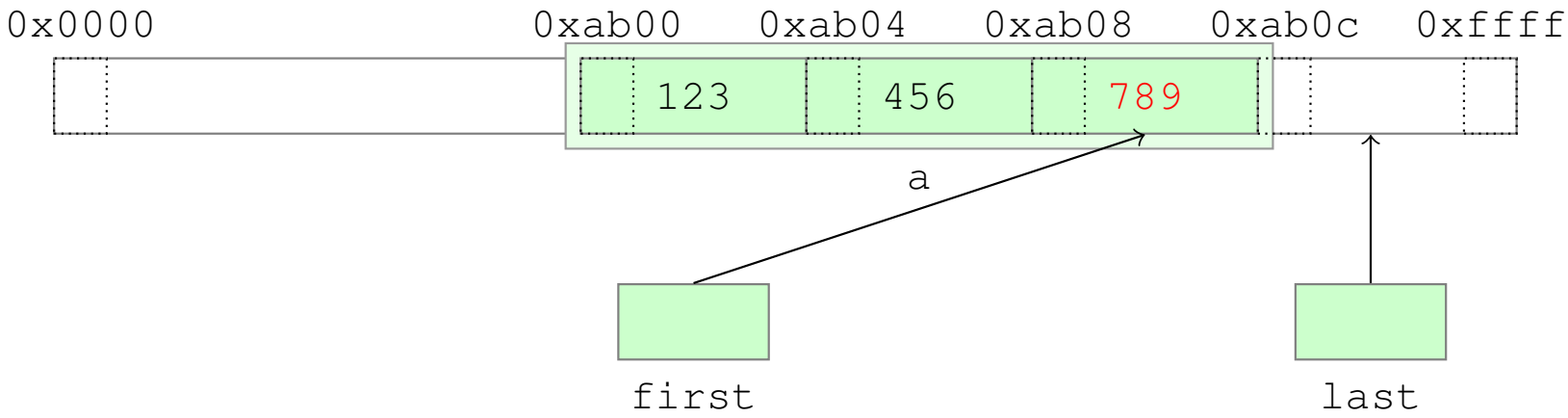

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



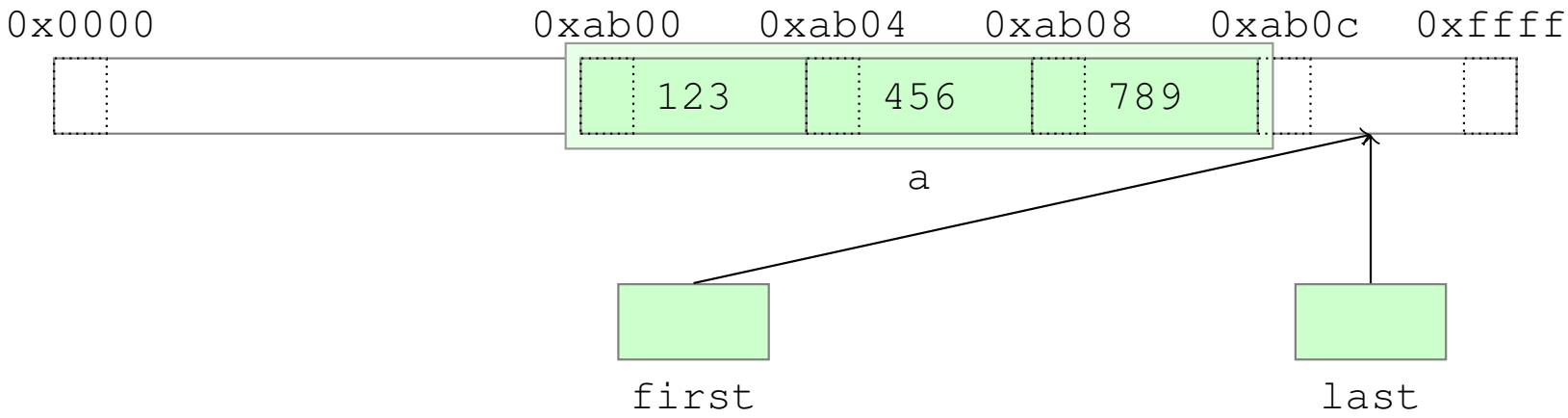

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



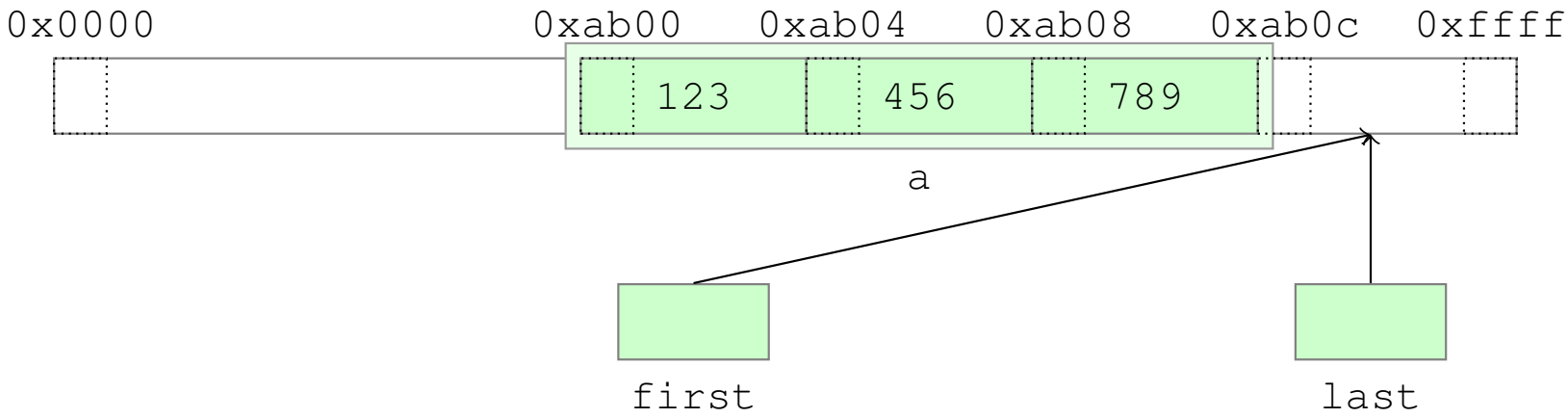

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



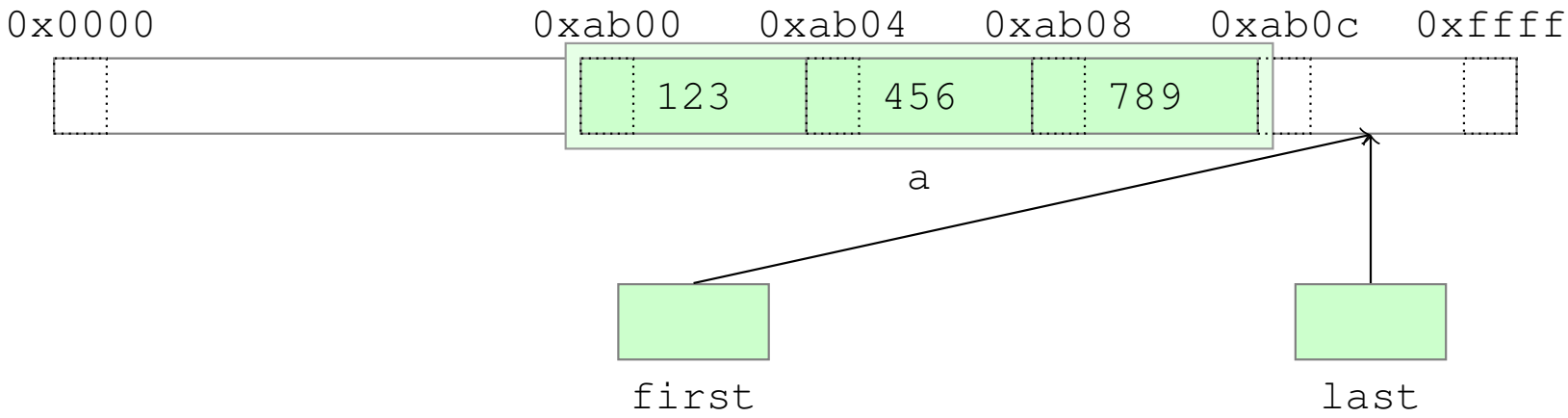

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)




```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)




```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin(); // or std::begin(a)
auto const last = a.end();    // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ dist(e); })
std::sort(a.begin(), a.end());
std::reverse(a.begin(), a.end());
if (std::find(a.begin(), a.end(), 123) == a.end()) {
  // 123 not found in a
}
std::rotate(a.begin(), a.begin() + 2, a.end());
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ dist(e); })
std::sort(a.begin(), a.end());
std::reverse(a.begin(), a.end());
if (std::find(a.begin(), a.end(), 123) == a.end()) {
  // 123 not found in a
}
std::rotate(a.begin(), a.begin() + 2, a.end());
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ dist(e); })
std::sort(a.begin(), a.end());
std::reverse(a.begin(), a.end());
if (std::find(a.begin(), a.end(), 123) == a.end()) {
  // 123 not found in a
}
std::rotate(a.begin(), a.begin() + 2, a.end());
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ dist(e); })
std::sort(a.begin(), a.end());
std::reverse(a.begin(), a.end());
if (std::find(a.begin(), a.end(), 123) == a.end()) {
  // 123 not found in a
}
std::rotate(a.begin(), a.begin() + 2, a.end());
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ dist(e); })
std::sort(a.begin(), a.end());
std::reverse(a.begin(), a.end());
if (std::find(a.begin(), a.end(), 123) == a.end()) {
  // 123 not found in a
}
std::rotate(a.begin(), a.begin() + 2, a.end());
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ dist(e); })
std::sort(a.begin(), a.end());
std::reverse(a.begin(), a.end());
if (std::find(a.begin(), a.end(), 123) == a.end()) {
  // 123 not found in a
}
std::rotate(a.begin(), a.begin() + 2, a.end());
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ dist(e); })
std::sort(a.begin(), a.end());
std::reverse(a.begin(), a.end());
if (std::find(a.begin(), a.end(), 123) == a.end()) {
  // 123 not found in a
}
std::rotate(a.begin(), a.begin() + 2, a.end());
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ dist(e); })
std::sort(a.begin(), a.end());
std::reverse(a.begin(), a.end());
if (std::find(a.begin(), a.end(), 123) == a.end()) {
  // 123 not found in a
}
std::rotate(a.begin(), a.begin() + 2, a.end());
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ dist(e); })
std::sort(a.begin(), a.end());
std::reverse(a.begin(), a.end());
if (std::find(a.begin(), a.end(), 123) == a.end()) {
  // 123 not found in a
}
std::rotate(a.begin(), a.begin() + 2, a.end());
```

std::vector

- Like an array, but adjusts its size dynamically
- Should be used as the *default* container

```
#include <vector>

std::vector<int> v; // size is 0
v.push_back(123);  // size is 1
v.push_back(456);  // size is 2
...
```

```
std::array<int,5> a;
std::vector<int> v(5); // initial size is 5, with 0 values
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(v.begin(), v.size(), [&](){ dist(e); })
std::sort(v.begin(), v.end());
std::reverse(v.begin(), v.end());
if (std::find(v.begin(), v.end(), 123) == v.end()) {
  // 123 not found in v
}
std::rotate(v.begin(), v.begin() + 2, v.end());
```

std::vector

- Like an array, but adjusts its size dynamically
- Should be used as the *default* container

```
#include <vector>

std::vector<int> v; // size is 0
v.push_back(123);  // size is 1
v.push_back(456);  // size is 2
...
```

```
std::array<int,5> a;
std::vector<int> v(5); // initial size is 5, with 0 values
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(v.begin(), v.size(), [&](){ dist(e); })
std::sort(v.begin(), v.end());
std::reverse(v.begin(), v.end());
if (std::find(v.begin(), v.end(), 123) == v.end()) {
  // 123 not found in v
}
std::rotate(v.begin(), v.begin() + 2, v.end());
```

std::vector

- Like an array, but adjusts its size dynamically
- Should be used as the *default* container

```
#include <vector>

std::vector<int> v; // size is 0
v.push_back(123);  // size is 1
v.push_back(456);  // size is 2
...
```

```
std::array<int,5> a;
std::vector<int> v(5); // initial size is 5, with 0 values
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(v.begin(), v.size(), [&](){ dist(e); })
std::sort(v.begin(), v.end());
std::reverse(v.begin(), v.end());
if (std::find(v.begin(), v.end(), 123) == v.end()) {
  // 123 not found in v
}
std::rotate(v.begin(), v.begin() + 2, v.end());
```

What is a C++ program

- A program consists of one or more translation units linked together
- A translation unit is the result of the compilation of a source (cpp) file with all its `#include`d headers
- A header contains declarations of C++ entities
- A program starts at a global function called `main`
  - `int main()`
  - `int main(int argc, char** argv)`

Example 1

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

// file: functions.h, function declarations

void print(double);
double rand();

#endif
```

```
// file: functions.cpp, function definitions

#include "functions.h"
// other #include's for the implementation of print and rand

void print(double d) { ... }
double rand() { ... }
```

Run the compiler to produce an object file

- `$ g++ ... -c functions.cpp`
- `functions.o`

Example 1

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

// file: functions.h, function declarations

void print(double);
double rand();

#endif
```

```
// file: functions.cpp, function definitions

#include "functions.h"
// other #include's for the implementation of print and rand

void print(double d) { ... }
double rand() { ... }
```

Run the compiler to produce an object file

- `$ g++ ... -c functions.cpp`
- `functions.o`

Example 1

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

// file: functions.h, function declarations

void print(double);
double rand();

#endif
```

```
// file: functions.cpp, function definitions

#include "functions.h"
// other #include's for the implementation of print and rand

void print(double d) { ... }
double rand() { ... }
```

Run the compiler to produce an object file

- `$ g++ ... -c functions.cpp`
- `functions.o`

Example 1

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

// file: functions.h, function declarations

void print(double);
double rand();

#endif
```

```
// file: functions.cpp, function definitions

#include "functions.h"
// other #include's for the implementation of print and rand

void print(double d) { ... }
double rand() { ... }
```

Run the compiler to produce an object file

- `$ g++ ... -c functions.cpp`
- `functions.o`

Example 1

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

// file: functions.h, function declarations

void print(double);
double rand();

#endif
```

```
// file: functions.cpp, function definitions

#include "functions.h"
// other #include's for the implementation of print and rand

void print(double d) { ... }
double rand() { ... }
```

Run the compiler to produce an object file

- `$ g++ ... -c functions.cpp`
- `functions.o`

Example 1

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

// file: functions.h, function declarations

void print(double);
double rand();

#endif
```

```
// file: functions.cpp, function definitions

#include "functions.h"
// other #include's for the implementation of print and rand

void print(double d) { ... }
double rand() { ... }
```

Run the compiler to produce an object file

- `$ g++ ... -c functions.cpp`
- `functions.o`

Example 1

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

// file: functions.h, function declarations

void print(double);
double rand();

#endif
```

```
// file: functions.cpp, function definitions

#include "functions.h"
// other #include's for the implementation of print and rand

void print(double d) { ... }
double rand() { ... }
```

Run the compiler to produce an object file

- `$ g++ ...  -c functions.cpp`
- `functions.o`

Example 1 (cont.)

```
// file: main.cpp, functions use

#include "functions.h"

void test_rand() { print(rand()); }

int main() {
  test_rand();
}
```

- Run the compiler to produce another object file
  - `$ g++ ... -c main.cpp`
  - `main.o`
- Run the linker to produce an executable from the object files
  - `$ g++ ... main.o functions.o`
  - `a.out`
- In most cases
  - `$ g++ ... main.cpp functions.cpp`
  - `a.out`
- Or use a build framework (make, cmake, scons, cmt, ...)

Example 1 (cont.)

```
// file: main.cpp, functions use

#include "functions.h"

void test_rand() { print(rand()); }

int main() {
  test_rand();
}
```

- Run the compiler to produce another object file
  - `$ g++ ... -c main.cpp`
  - `main.o`
- Run the linker to produce an executable from the object files
  - `$ g++ ... main.o functions.o`
  - `a.out`
- In most cases
  - `$ g++ ... main.cpp functions.cpp`
  - `a.out`
- Or use a build framework (make, cmake, scons, cmt, ...)

Example 1 (cont.)

```
// file: main.cpp, functions use

#include "functions.h"

void test_rand() { print(rand()); }

int main() {
  test_rand();
}
```

- Run the compiler to produce another object file
  - `$ g++ ... -c main.cpp`
  - `main.o`
- Run the linker to produce an executable from the object files
  - `$ g++ ... main.o functions.o`
  - `a.out`
- In most cases
  - `$ g++ ... main.cpp functions.cpp`
  - `a.out`
- Or use a build framework (make, cmake, scons, cmt, ...)

Example 1 (cont.)

```
// file: main.cpp, functions use

#include "functions.h"

void test_rand() { print(rand()); }

int main() {
  test_rand();
}
```

- Run the compiler to produce another object file
  - `$ g++ ...  -c main.cpp`
  - `main.o`
- Run the linker to produce an executable from the object files
  - `$ g++ ...  main.o functions.o`
  - `a.out`
- In most cases
  - `$ g++ ...  main.cpp functions.cpp`
  - `a.out`
- Or use a build framework (make, cmake, scons, cmt, ...)

Example 1 (cont.)

```
// file: main.cpp, functions use

#include "functions.h"

void test_rand() { print(rand()); }

int main() {
  test_rand();
}
```

- Run the compiler to produce another object file
  - `$ g++ ... -c main.cpp`
  - `main.o`
- Run the linker to produce an executable from the object files
  - `$ g++ ... main.o functions.o`
  - `a.out`
- In most cases
  - `$ g++ ... main.cpp functions.cpp`
  - `a.out`
- Or use a build framework (make, cmake, scons, cmt, ...)

Example 1 (cont.)

```
// file: main.cpp, functions use

#include "functions.h"

void test_rand() { print(rand()); }

int main() {
  test_rand();
}
```

- Run the compiler to produce another object file
  - `$ g++ ... -c main.cpp`
  - `main.o`
- Run the linker to produce an executable from the object files
  - `$ g++ ... main.o functions.o`
  - `a.out`
- In most cases
  - `$ g++ ... main.cpp functions.cpp`
  - `a.out`
- Or use a build framework (make, cmake, scons, cmt, ...)

Example 1 (cont.)

```
// file: main.cpp, functions use

#include "functions.h"

void test_rand() { print(rand()); }

int main() {
  test_rand();
}
```

- Run the compiler to produce another object file
  - `$ g++ ... -c main.cpp`
  - `main.o`
- Run the linker to produce an executable from the object files
  - `$ g++ ... main.o functions.o`
  - `a.out`
- In most cases
  - `$ g++ ... main.cpp functions.cpp`
  - `a.out`
- Or use a build framework (make, cmake, scons, cmt, ...)

Example 2

```
#ifndef COMPLEX_HPP
#define COMPLEX_HPP

// file: complex.hpp

class Complex {
  double r;
  double i;
 public:
  Complex(double x = 0., double y = 0.);
  double real() const;
  double imag() const;
  Complex& operator+=(Complex const& o);
  ...
};

#endif
```

- Only method's declarations, not their definition

+ Better physical decoupling: the code can change without the need to recompile the clients

- Worse performance: the compiler cannot inline

Example 2

```
#ifndef COMPLEX_HPP
#define COMPLEX_HPP

// file: complex.hpp

class Complex {
  double r;
  double i;
 public:
  Complex(double x = 0., double y = 0.);
  double real() const;
  double imag() const;
  Complex& operator+=(Complex const& o);
  ...
};

#endif
```

- Only method's declarations, not their definition
- \+ Better physical decoupling: the code can change without the need to recompile the clients
- \- Worse performance: the compiler cannot inline

Example 2

```
#ifndef COMPLEX_HPP
#define COMPLEX_HPP

// file: complex.hpp

class Complex {
  double r;
  double i;
 public:
  Complex(double x = 0., double y = 0.);
  double real() const;
  double imag() const;
  Complex& operator+=(Complex const& o);
  ...
};

#endif
```

- Only method's declarations, not their definition
- \+ Better physical decoupling: the code can change without the need to recompile the clients
- \- Worse performance: the compiler cannot inline

Example 2 (cont.)

```
#ifndef COMPLEX_FUNCTIONS_HPP
#define COMPLEX_FUNCTIONS_HPP

// file: complex_functions.hpp

#include "complex.hpp"

inline bool operator==(Complex const& lhs, Complex const& rhs) {
  return lhs.r == rhs.r && lhs.i == rhs.i;
}

Complex operator+(Complex const& lhs, Complex const& rhs);

double norm2(Complex const& c);

...
```

Functions can be defined in a header file, but they need to be declared inline to prevent multiple definitions

Example 2 (cont.)

```
#ifndef COMPLEX_FUNCTIONS_HPP
#define COMPLEX_FUNCTIONS_HPP

// file: complex_functions.hpp

#include "complex.hpp"

inline bool operator==(Complex const& lhs, Complex const& rhs) {
  return lhs.r == rhs.r && lhs.i == rhs.i;
}

Complex operator+(Complex const& lhs, Complex const& rhs);

double norm2(Complex const& c);

...
```

Functions can be defined in a header file, but they need to be declared inline to prevent multiple definitions

Example 2 (cont.)

```
// file: complex.cpp

#include "complex.hpp"

Complex::Complex(double x, double y) : r{x}, i{y} {}

double Complex::real() const { return r; }

double Complex::imag() const { return i; }

Complex& Complex::operator+=(Complex const& o) {
  r += o.r;
  i += o.i;
  return *this;
}

...
```

Run the compiler to produce an object file

- `$ g++ ...  -c complex.cpp`
- `complex.o`

Example 2 (cont.)

```
// file: complex_functions.cpp

#include "complex_functions.hpp"
#include "complex.hpp"

// operator==() already defined in the header file

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs;
  return result += rhs;
}

double norm2(Complex const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}

...
```

- Better include `complex.hpp` explicitly
  - include guards protect from multiple inclusions
- Run the compiler to produce an object file
  - `$ g++ ... -c complex_functions.cpp`

Example 2 (cont.)

```
// file: complex_functions.cpp

#include "complex_functions.hpp"
#include "complex.hpp"

// operator==() already defined in the header file

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs;
  return result += rhs;
}

double norm2(Complex const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}

...
```

- Better include `complex.hpp` explicitly
  - include guards protect from multiple inclusions
- Run the compiler to produce an object file
  - `$ g++ ... -c complex_functions.cpp`

Example 2 (cont.)

```
// file: complex_functions.cpp

#include "complex_functions.hpp"
#include "complex.hpp"

// operator==() already defined in the header file

Complex operator+(Complex const& lhs, Complex const& rhs) {
  auto result = lhs;
  return result += rhs;
}

double norm2(Complex const& c) {
  return c.real() * c.real() + c.imag() * c.imag();
}

...
```

- Better include `complex.hpp` explicitly
  - include guards protect from multiple inclusions
- Run the compiler to produce an object file
  - `$ g++ ... -c complex_functions.cpp`

Example 2 (cont.)

```
// file: main.cpp

#include "complex.hpp"
#include "complex_functions.hpp"

int main() {
  Complex c1{1., 2.};
  Complex c2{3.};
  Complex c3 = c1 + c2;
  norm2(c3);
}
```

- Run the compiler to produce an object file
  - `$ g++ ... -c main.cpp`
- Run the linker to produce an executable
  - `$ g++ main.o complex_functions.o complex.o`
- Or do all in one step
  - `$ g++ main.cpp complex_functions.cpp complex.cpp`

Example 2 (cont.)

```
// file: main.cpp

#include "complex.hpp"
#include "complex_functions.hpp"

int main() {
  Complex c1{1., 2.};
  Complex c2{3.};
  Complex c3 = c1 + c2;
  norm2(c3);
}
```

- Run the compiler to produce an object file
  - `$ g++ ...  -c main.cpp`
- Run the linker to produce an executable
  - `$ g++ main.o complex_functions.o complex.o`
- Or do all in one step
  - `$ g++ main.cpp complex_functions.cpp complex.cpp`

Example 3

```
#ifndef MATHLIB_COMPLEX_HPP
#define MATHLIB_COMPLEX_HPP

// file: mathlib/complex.hpp

namespace mathlib {
  template<typename FP>
  class Complex {
    FP r;
    FP i;
   public:
    Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
    auto real() const { return r; }
    auto imag() const { return i; }
    Complex& operator+=(Complex const& o)
    { ... }
    ...
  };
}

#endif
```

Example 3 (cont.)

```
#ifndef MATHLIB_FUNCTIONS_HPP
#define MATHLIB_FUNCTIONS_HPP

// file: mathlib/complex_functions.hpp

#include "mathlib/complex.hpp"

namespace mathlib {

  template<class FP>
  auto norm2(Complex<FP> const& c) { ... }

  template<class FP>
  Complex<FP>
  operator+(Complex<FP> const& lhs, Complex<FP> const& rhs)
  { ... }

  template<class FP>
  bool
  operator==(Complex<FP> const& lhs, Complex<FP> const& rhs)
  { ... }

  ...
}
```

Example 3 (cont.)

- No corresponding `cpp` file for `complex.hpp` and `complex_functions.hpp`
- The function implementation code should be visible to the compiler when compiling the clients

```
// file: main.cpp

#include "mathlib/complex.hpp"
#include "mathlib/complex_functions.hpp"

int main() {
  namespace ml = mathlib;
  ml::Complex<double> c1{1., 2.};
  ml::Complex<double> c2{3.};
  ml::Complex<double> c3 = c1 + c2;
  norm2(c3);
}
```

In this case there is only one translation unit

- `$ g++ main.cpp`

Example 3 (cont.)

- No corresponding `cpp` file for `complex.hpp` and `complex_functions.hpp`
- The function implementation code should be visible to the compiler when compiling the clients

```
// file: main.cpp

#include "mathlib/complex.hpp"
#include "mathlib/complex_functions.hpp"

int main() {
  namespace ml = mathlib;
  ml::Complex<double> c1{1., 2.};
  ml::Complex<double> c2{3.};
  ml::Complex<double> c3 = c1 + c2;
  norm2(c3);
}
```

In this case there is only one translation unit

- `$ g++ main.cpp`

Example 3 (cont.)

- No corresponding `cpp` file for `complex.hpp` and `complex_functions.hpp`
- The function implementation code should be visible to the compiler when compiling the clients

```
// file: main.cpp

#include "mathlib/complex.hpp"
#include "mathlib/complex_functions.hpp"

int main() {
  namespace ml = mathlib;
  ml::Complex<double> c1{1., 2.};
  ml::Complex<double> c2{3.};
  ml::Complex<double> c3 = c1 + c2;
  norm2(c3);
}
```

In this case there is only one translation unit

- `$ g++ main.cpp`