References, Arrays, and Pointers

Overview

So far, we have mostly worked with *fundamental types*

- `void`
- Arithmetic types such as `int`, `float`, etc.

Much of the power of C++ comes from the ability to define *compound types*

- Functions
- Classes (covered in the next lecture)
- References
- Arrays
- Pointers

Reference Declaration (1)

A reference declaration declares an alias to an already-existing object or function

- **Lvalue reference**: *type*`&` *declarator*
- **Rvalue reference**: *type*`&&` *declarator*
- Most of the time, *declarator* will simply be a name

References have some peculiarities

- There are no references to `void`
- References are immutable (although the referenced object may be mutable)
- References are not objects, i.e. they do not necessarily occupy storage

Since references are not objects

- There are no references or pointers to references
- There are no arrays of references

Reference Declaration (2)

The & or && tokens are part of the *declarator*, not the type

```
int i = 10;
int& j = i, k = i;  // j is reference to int, k is int
```

However, we may omit or insert whitespaces before and after the & or && tokens

- Both `int& j = i;` and `int &j = i;` are valid C++
- By convention, we use the former notation (`int& j = i;`)
- To avoid confusion, statements should declare only one identifier at a time
- Very rarely, exceptions to this rule are necessary in the *init-statements* of `if` and `switch` statements as well as `for` loops

Reference Initialization

Definitions of references to a type T must be initialized to refer to a valid object or function

- An object of type T
- A function of type T
- An object implicitly convertible to T

Declarations of references do not need intializers:

- Function parameter declarations
- Function return type declarations
- Class member declarations
- With the `extern` specifier

Lvalue References (1)

As an alias for existing objects

```
unsigned i = 10;
unsigned j = 42;
unsigned& r = i;  // r is an alias for i

r = 21;           // modifies i to be 21
r = j;            // modifies i to be 42

i = 123;
j = r;            // modifies j to be 123
```

Lvalue References (2)

To implement pass-by-reference semantics for function calls

```
void foo(int& value) {
    value += 42;
}

int main() {
    int i = 10;
    foo(i);      // i == 52
    foo(i);      // i == 94
}
```

Lvalue References (3)

To turn a function call into an lvalue expression

```
int global0 = 0;
int global1 = 0;

int& foo(unsigned which) {
    if (!which)
        return global0;
    else
        return global1;
}

int main() {
    foo(0) = 42;  // global0 == 42
    foo(1) = 14;  // global1 == 14
}
```

Rvalue References (1)

Can not (directly) bind to lvalues

```
int i = 10;
int&& j = i;  // ERROR: Cannot bind rvalue reference to lvalue
int&& k = 42; // OK
```

Extend the lifetimes of temporary objects

```
int i = 10;
int j = 32;

int&& k = i + j;  // k == 42
k += 42;          // k == 84;
```

Rvalue References (2)

Allow overload resolution to distinguish between lvalues and rvalues

```
void foo(int& x);
void foo(const int& x);
void foo(int&& x);

int& bar();
int baz();

int main() {
    int i = 42;
    const int j = 84;

    foo(i);     // calls foo(int&)
    foo(j);     // calls foo(const int&)
    foo(123);   // calls foo(int&&)

    foo(bar())  // calls foo(int&)
    foo(baz())  // calls foo(int&&)
}
```

References and CV-Qualifiers

References themselves cannot be cv-qualified

- However, the referenced type may be cv-qualified
- A reference to T can be initialized from a type that is less cv-qualified than T (e.g. `const int&` can be initialized from `int&`)

```
int i = 10;
const int& j = i;
int& k = j;  // ERROR: binding reference of type int& to
             //        const int discards cv-qualifiers
j = 42;      // ERROR: assignment of read-only reference
```

Lvalue references to `const` also extend the lifetime of temporary objects

```
int i = 10;
int j = 32;
const int& k = i + j;  // OK, but k is immutable
```

Dangling references

It is possible to write programs where the lifetime of a referenced object ends while references to it still exist.

- This can already happen when referencing objects with automatic storage duration
- Results in *dangling reference* and undefined behavior

Example

```
int& foo() {
    int i = 42;
    return i;    // ERROR: Returns dangling reference
}
```

Array Declaration (1)

An array declaration declares an object of array type (also: C-style array)

- *type declarator*[*expression*]
- *expression* must be an expression which evaluates to an integral constant at **compile time**
- Again, due to weird parsing rules [*expression*] is part of the declarator
- *type*[*expression*] can be used as a type outside of declarators

For example: `T a[N];` for some type `T` and compile-time constant `N`

- a consists of `N` contiguously allocated elements of type `T`
- Elements are numbered `0`, ..., `N - 1`
- Elements can be accessed with the subscript operator `[]`, e.g. `a[0]`, ..., `a[N - 1]`
- Without an initializer, every element of a is uninitialized

Array Declaration (2)

Example

```
unsigned short a[10];

for (unsigned i = 0; i < 10; ++i)
    a[i] = i + 1;
```

Array objects are lvalues, but they cannot be assigned to

```
unsigned short a[10];
unsigned short b[10];

a = b;  // ERROR: a is an array
```

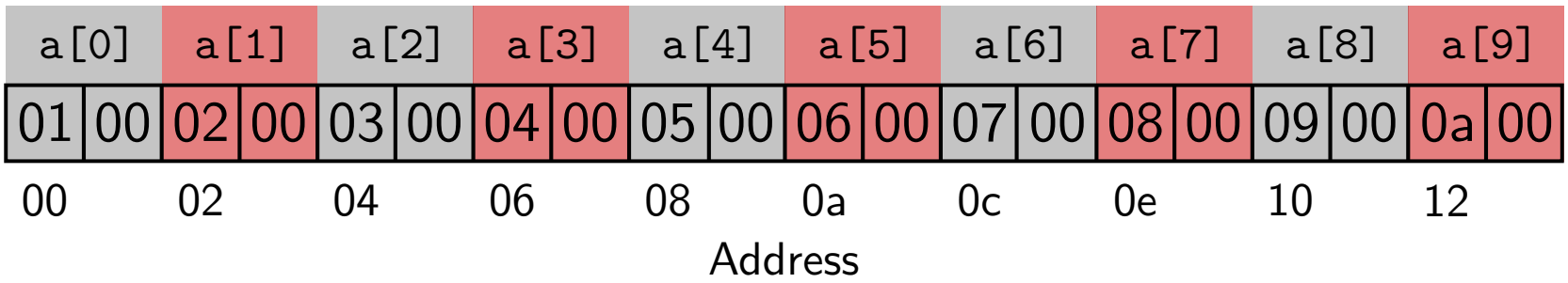
Arrays cannot be returned from functions

```
int[] foo();  // ERROR
```

Array Declaration (3)

Elements of an array are allocated **contiguously** in memory

- Given `unsigned short a[10];` containing the integers `1` through `10`
- Assuming a 2-byte `unsigned short` type
- Assuming little-endian byte ordering



Arrays are just dumb chunks of memory

- Out-of-bounds accesses are not automatically detected, and do not necessarily lead to a crash
- May lead to rather weird bugs
- Exist mainly due to compatibility requirements with C

Array Declaration (4)

The elements of an array can be arrays themselves

```
unsigned short b[3][2];

for (unsigned i = 0; i < 3; ++i)
    for (unsigned j = 0; j < 2; ++j)
        b[i][j] = 3 * i + j;
```

Elements are still allocated contiguously in memory

- b can be thought of as a 3×2 matrix in row-major format
- The subscript operator simply returns an array object on the first level, to which the subscript operator can be applied again (`(b[i])[j]`)

Array Initialization

Arrays can be default-initialized, in which case every element is default-initialized

```
unsigned short a[10] = {};  // a contains 10 zeros
```

Arrays can be list-initialized, in which case the size may be omitted

```
unsigned short a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Multi-dimensional arrays may also be list-initialized, but only the first dimension may have unknown bound

```
unsigned short b[][2] = {
    {0, 1},
    {2, 3},
    {4, 5}
}
```

size_t

C++ has a designated type for indexes and sizes: `std::size_t` from `<cstddef>`

- `size_t` is an unsigned integer type that is large enough to represent sizes and all possible array indexes on the target architecture
- The C++ language and standard library use `size_t` when handling indexes or sizes
- Generally, use `size_t` for array indexes and sizes
- Sometimes you can also use smaller integer types (e.g. `unsigned`) when working with small arrays

std::array

C-style arrays should be avoided whenever possible

- Use the `std::array` type defined in the `<array>` standard header instead
- Same semantics as a C-style array
- Optional bounds-checking and other useful features
- `std::array` is a *template type* with two template parameters (the element type and count)

Example

```
#include <array>

int main() {
    std::array<unsigned short, 10> a;
    for (size_t i = 0; i < a.size(); ++i)
        a[i] = i + 1;  // no bounds checking
}
```

std::vector (1)

`std::array` is inflexible due to compile-time fixed size

- The `std::vector` type defined in the `<vector>` standard header provides dynamically-sized arrays
- Storage is automatically expanded and contracted as needed
- Elements are still stored contiguously in memory

Useful functions

- `push_back` – inserts an element at the end of the vector
- `size` – queries the current size
- `clear` – clears the contents
- `resize` – change the number of stored elements
- The subscript operator can be used with similar semantics as for C-style arrays

Familiarize yourself with the reference documentation on `std::vector`

std::vector (2)

Example

```
#include <iostream>
#include <vector>

int main() {
    std::vector<unsigned short> a;
    for (size_t i = 0; i < 10; ++i)
        a.push_back(i + 1);

    std::cout << a.size() << std::endl;  // prints 10
    a.clear();
    std::cout << a.size() << std::endl;  // prints 0

    a.resize(10);  // a now contains 10 zeros
    std::cout << a.size() << std::endl;  // prints 10

    for (unsigned i = 0; i < 10; ++i)
        a[i] = i + 1;
}
```

Range-For (1)

Execute a `for`-loop over a range

```
for (init-statement; range-declaration : range-expression)
    loop-statement
```

Explanation

- Executes *`init-statement`* once, then executes *`loop-statement`* once for each element in the range defined by *`range-expression`*
- *`range-expression`* may be an expression that represents a sequence (e.g. an array or an object for which `begin` and `end` functions are defined, such as `std::vector`)
- *`range-declaration`* should declare a named variable of the element type of the sequence, or a reference to that type
- *`init-statement`* may be omitted

Range-For (2)

Example

```
#include <iostream>
#include <vector>

int main() {
    std::vector<unsigned short> a;

    // no range-for, we need the index
    for (size_t i = 0; i < 10; ++i)
        a.push_back(i + 1);

    // range-for
    for (const unsigned short& e : a)
        std::cout << e << std::endl;
}
```

Storage of Objects

A "region of storage" has a physical equivalent

- Typically, objects reside in main memory, either on the stack or on the heap
- Up to now (and for some lectures to come), we have almost exclusively dealt with objects residing on the stack

Objects reside at some specific *location* in main memory

- As we have seen in the first lecture, this location can be identified by an *address* in main memory
- It is convenient to think of addresses as simple offsets from the beginning of the address space
- Pointers are a feature of C++ to obtain and interact with these addresses

Pointer Declaration (1)

A pointer declaration declares a variable of pointer type

- *type*`*` *cv declarator*
- *declarator* may be any other declarator, except for a reference declarator
- *cv* specifies the cv-qualifiers of the pointer (not the pointed-to type), and may be omitted
- Analogous to reference declarations, the `*` token is part of the declarator, not the type

Notes

- A pointer to an object represents the address of the *first* byte in memory that is occupied by that object
- As opposed to references, pointers are themselves objects
- Consequently, pointers to pointers are allowed

Pointer Declaration (2)

Examples of valid pointer declarations

```
int* a;                 // pointer to int
const int* b;           // pointer to const int
int* const c;           // const pointer to int
const int* const d;     // const pointer to const int
```

Pointer-to-pointer declarations

```
int** e;                          // pointer to pointer to int
const int* const* const f;        // const pointer to const pointer
                                  // to const int
```

Contraptions like the declaration of `f` are very rarely (if at all) necessary

The Address-Of Operator

In general, there exists no implicit conversion from a pointed-to type (e.g. `int`) to its pointer type (e.g. `int*`)

- In order to obtain a pointer to an object, the built-in unary address-of operator & has to be used
- Given an lvalue expression a, &a returns a pointer to the value of the expression
- The cv-qualification of a is retained

Example

```
int a = 10;
const int b = 42;
int* c = &a;          // OK: c points to a
const int* d = &b;    // OK: d points to b
int* e = &b;          // ERROR: invalid conversion from
                      // const int* to int*
```

The Indirection Operator

In general, there exists no implicit conversion from a pointer type (e.g. `int*`) to its pointed-to type (e.g. `int`)

- In order to access the pointed-to object, the built-in unary indirection operator `*` has to be used
- Given an expression *expr* of pointer type, **expr* returns an lvalue reference to the pointed-to object
- The cv-qualifiers of the pointed-to type are retained
- Applying the indirection operator is also called *dereferencing* a pointer

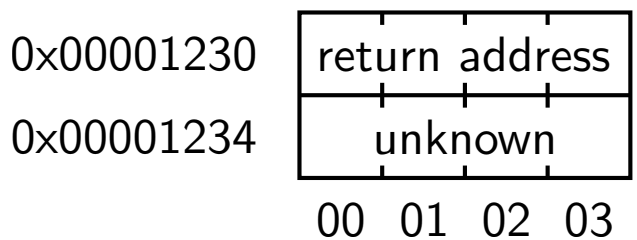
Example

```
int a = 10;
int* c = &a;
int& d = *c;  // reference to a
d = 123;      // a == 123
*c = 42;      // a == 42
```

What is Happening? (1)

```
int main() {



}
```

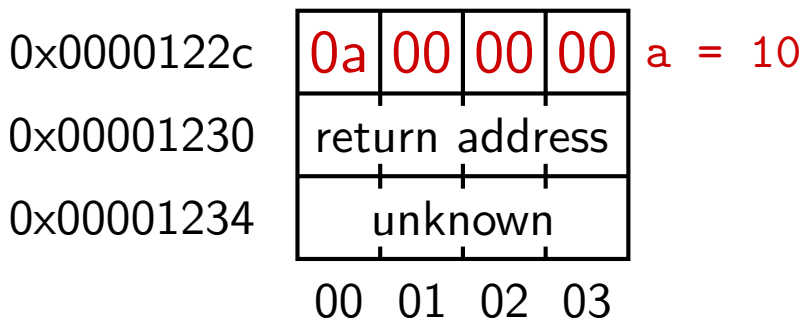
Stack Memory



What is Happening? (2)

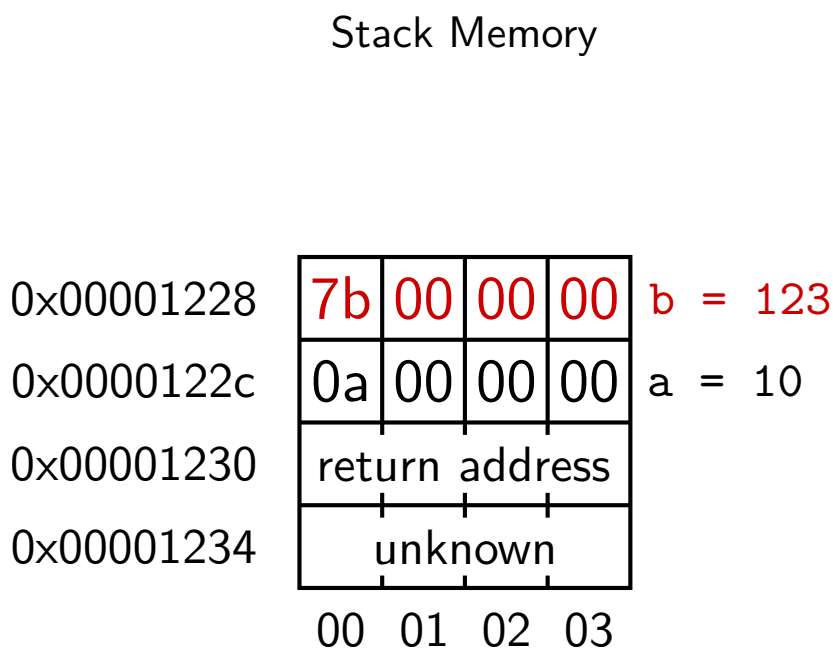
```
int main() {
    int a = 10;



}
```

Stack Memory



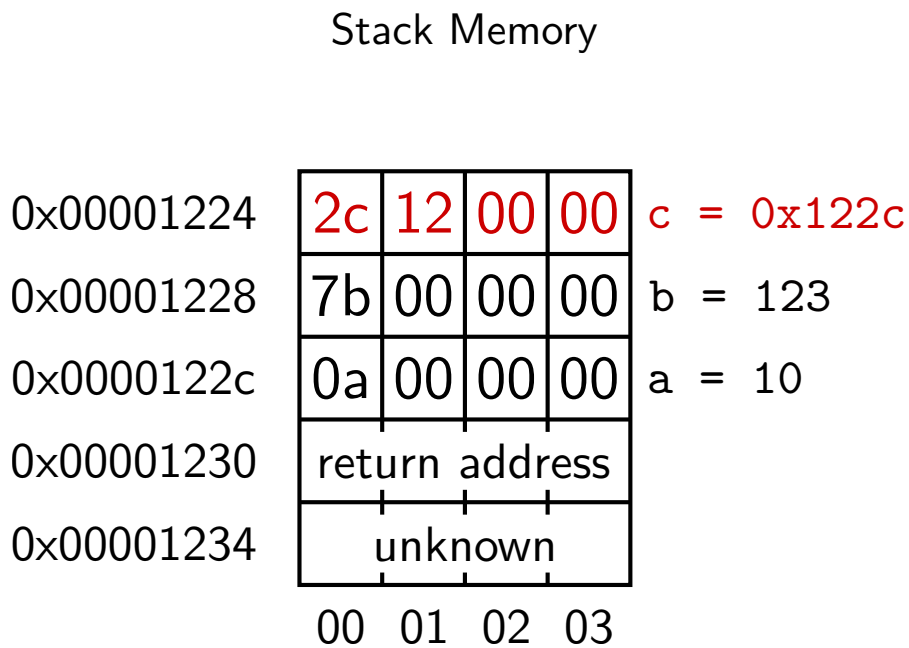
What is Happening? (3)

```
int main() {
    int a = 10;
    int b = 123;




}
```



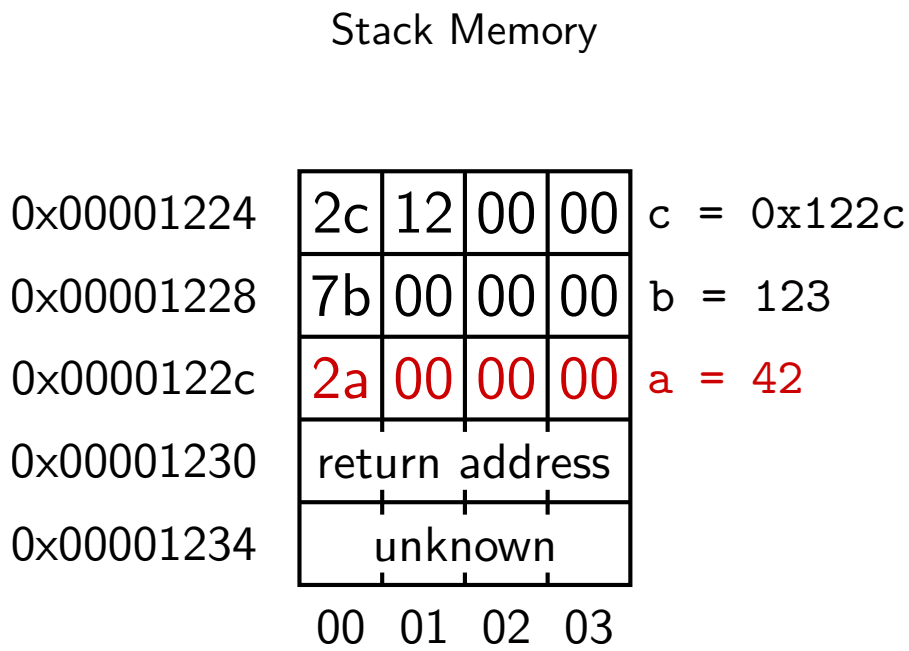
What is Happening? (4)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;




}
```



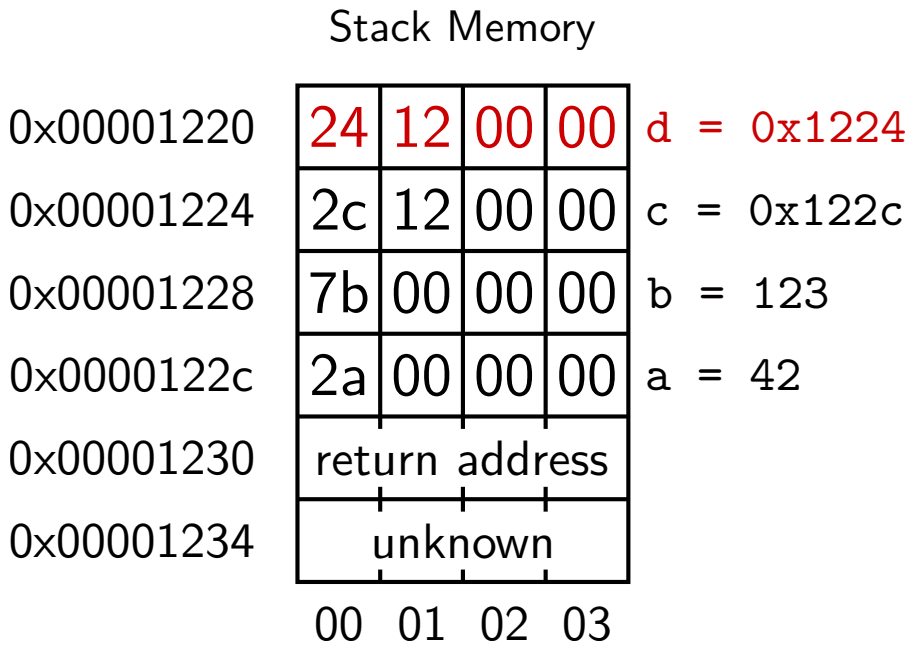
What is Happening? (5)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;



}
```



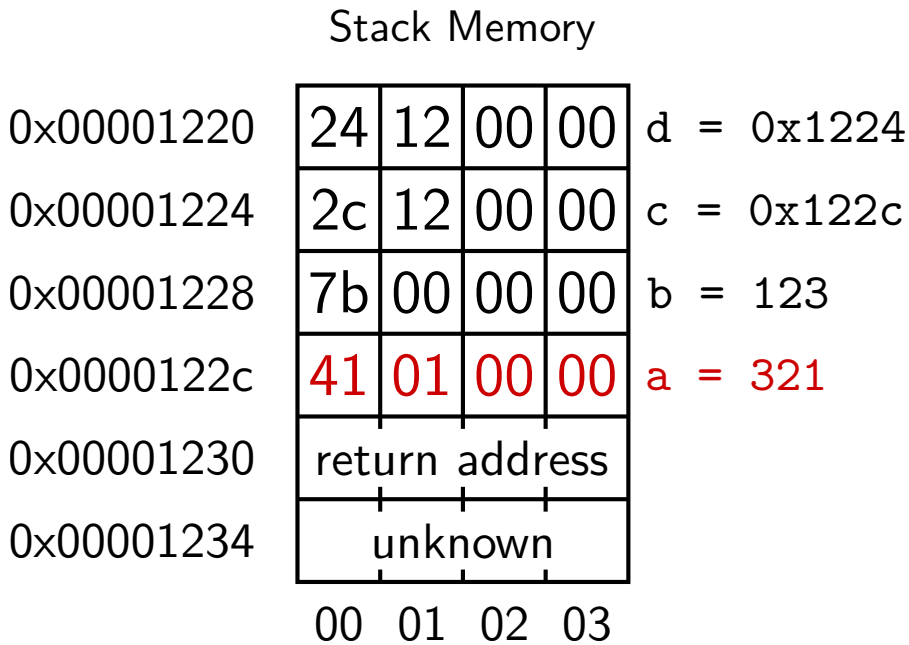
What is Happening? (6)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;



}
```



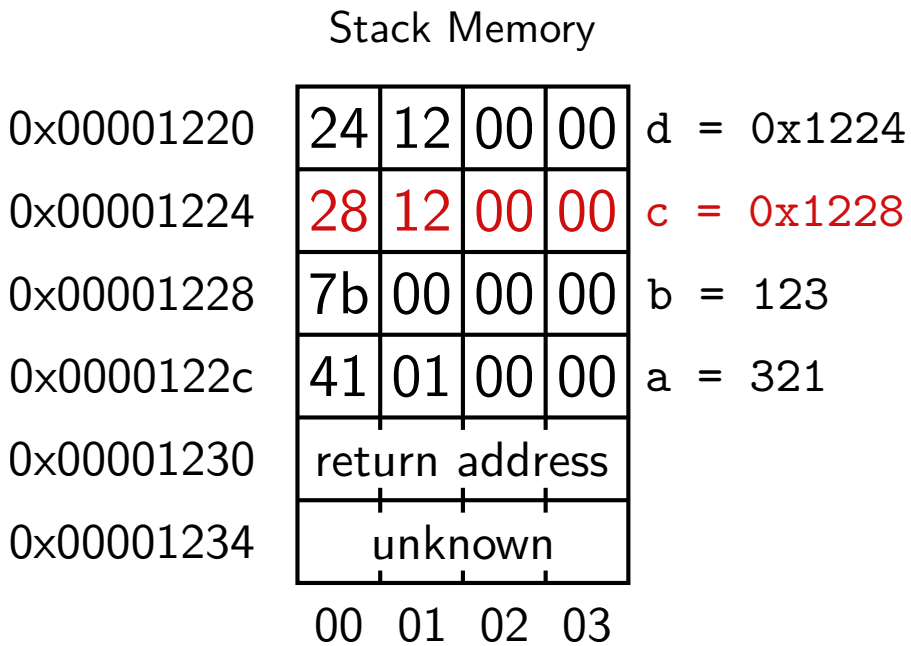
What is Happening? (7)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;




}
```



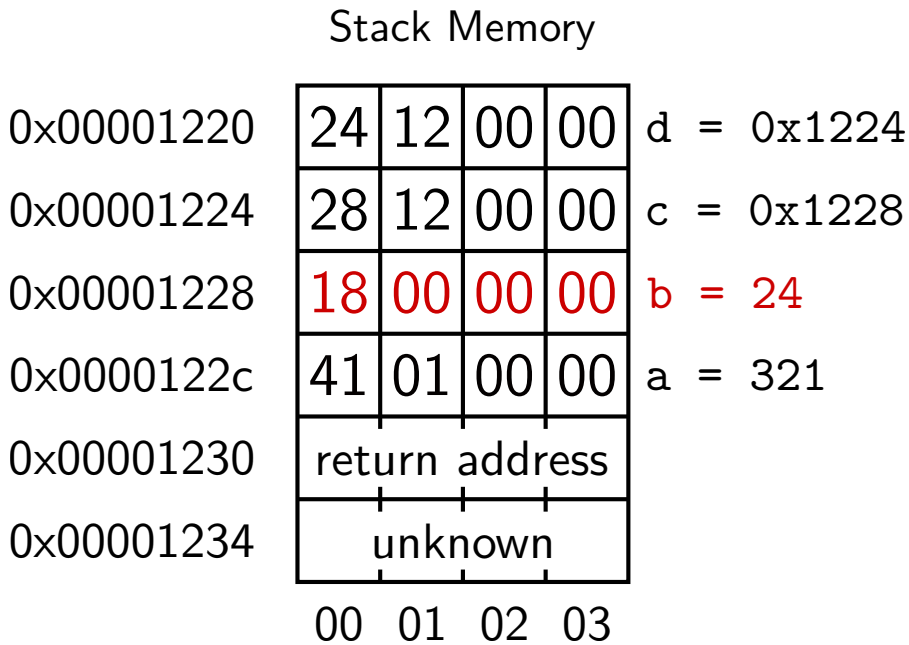
What is Happening? (8)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;
    *d = &b;



}
```



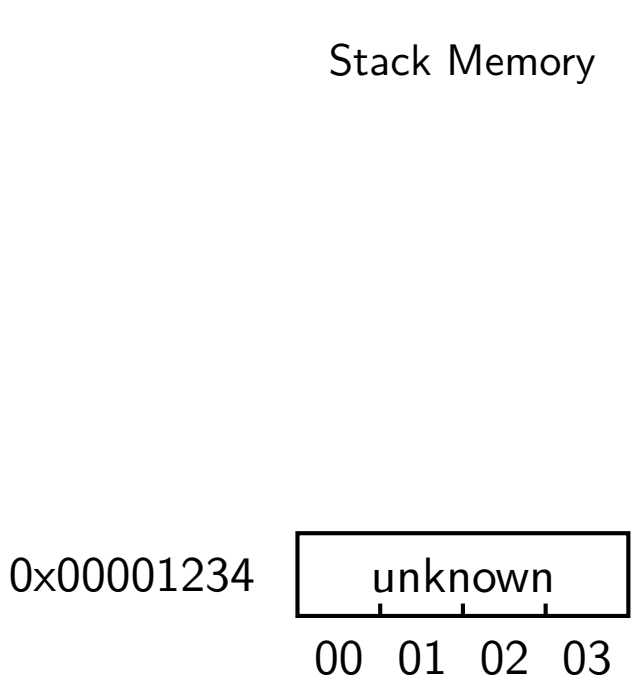
What is Happening? (9)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;
    *d = &b;
    **d = 24;


}
```



What is Happening? (10)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;
    *d = &b;
    **d = 24;

    return 0;
}
```



Null Pointers

A pointer may not point to any object at all

- Indicated by the special value and corresponding literal `nullptr`
- Pointers of the same type which are both null pointers are considered equal
- It is undefined behavior to dereference a null pointer

Undefined behavior can lead to surprising results

foo.cpp

```
int foo(const int* ptr) {
    int v = *ptr;

    if (ptr == nullptr)
        return 42;

    return v;
}
```

foo.o

```
foo(int*):
    movl    (%rdi), %eax
    ret
```

Array to Pointer Decay

Arrays and pointers have many similarities

- There is an implicit conversion from values of array type to values of pointer type
- The conversion constructs a pointer to the first element of an array
- The pointer type must be at least as cv-qualified as the array type

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr = array;

    std::cout << "The first element of array is ";
    std::cout << *ptr << std::endl;
}
```

The Subscript Operator

The subscript operator is defined on pointer types

- Treats the pointer as a pointer to the first element of an array
- Follows the same semantics as the subscript operator on array types

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr = array;

    std::cout << "The elements of array are";
    for (unsigned i = 0; i < 3; ++i)
        std::cout << " " << ptr[i];
    std::cout << std::endl;
}
```

Special Case: String Literals

String literals are another artifact of C compatibility

- String literals are immutable null-terminated character arrays
- That is, the type of a string literal with `N` characters is `const char[N + 1]`
- Most of the time, programmers take advantage of array-to-pointer decay and write `const char* str = "foo";`
- The character type can be controlled by the prefixes known from character literals (i.e. `u8"string"`, `u"string"`, or `U"string"`)

C-style strings should never be used!

- The C++ standard library provides the much safer `std::string` and `std::string_view` types
- Unfortunately, libraries or syscalls often require C-style string parameters
- If required, the standard library types can expose the C-style string representation

Arithmetic on Pointers (1)

Some arithmetic operators are defined between pointers and integral types

- Treats the pointer as a pointer to some element of an array
- Adding `i` to a pointer moves the it `i` elements to the right
- Subtracting `i` from a pointer moves it `i` elements to the left
- In general, for a pointer p the expressions `p[i]` and `*(p + i)` are equivalent

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr = &array[1];

    std::cout << "The previous element is ";
    std::cout << *(ptr - 1) << std::endl;
    std::cout << "The next element is ";
    std::cout << *(ptr + 1) << std::endl;
}
```

Arithmetic on Pointers (2)

Special care has to be taken to only dereference valid pointers

- Especially important since it is valid to take the *past-the-end* pointer of an array or `std::vector`

Example

```
int main() {
    std::vector<int> v;
    v.resize(10);

    const int* firstPtr = &v[0];  // OK: valid pointer
    const int* lastPtr = &v[10];  // OK: past-the-end pointer

    int last1 = *lastPtr;   // ERROR, might segfault
    int last2 = v[10];      // ERROR, might segfault
}
```

Arithmetic on Pointers (3)

Subtraction is defined between pointers

- Treats both pointers as pointers to some elements of an array
- Computes the number of elements between these two pointers

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr1 = &array[0];
    const int* ptr2 = &array[3];  // past-the-end pointer

    std::cout << "There are " << (ptr2 - ptr1) << " elements ";
    std::cout << "in array" << std::endl;
}
```

Comparisons on Pointers

The comparison operators are defined between pointers

- Interprets the addresses represented by the pointers as integers and compares them
- Only defined if the pointers point to elements of the same array

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};

    std::cout << "The elements of array are"
    for (const int* it = &array[0]; it < &array[3]; ++it)
        std::cout << " " << *it;
    std::cout << std::endl;
}
```

Void Pointers

Pointers to `void` are allowed

- A pointer to an object of any type can implicitly be converted to a pointer to `void`
- The void pointer must be at least as cv-qualified as the original pointer
- The pointer value (i.e. the address) is unchanged

Usage

- Used to pass objects of unknown type
- Extensively used in C interfaces (e.g. `malloc`, `qsort`, ...)
- Only few operations are defined on void pointers (mainly assignment)
- In order to use the pointed-to object, one must *cast* the void pointer to the required type

static_cast (1)

The `static_cast` conversion is used to cast between related types

```
static_cast< new_type > ( expression )
```

Explanation

- Converts the value of *expression* to a value of *new_type*
- *new_type* must be at least as cv-qualified as the type of *expression*
- Can be used to convert void pointers to pointers of another type
- Many more use cases (see reference documentation)

static_cast (2)

Void pointers

```
int i = 42;
void* vp = &i;
int* ip = static_cast<int*>(vp);
```

Other related types

```
int sum(int a, int b);
double sum(double a, double b);

int main() {
    int a = 42;
    double b = 3.14;

    double x = sum(a, b);                          // ERROR: ambiguous
    double y = sum(static_cast<double>(a), b);     // OK
    int z = sum(a, static_cast<int>(b));           // OK
}
```

reinterpret_cast

The `reinterpret_cast` conversion is used to convert between unrelated types

```
reinterpret_cast < new_type > ( expression )
```

Explanation

- Interprets the underlying bit pattern of the value of *expression* as a value of *new_type*
- *new_type* must be at least as cv-qualified as the type of *expression*
- Usually does not generate any CPU instructions

Only a very restricted set of conversions is allowed

- A pointer to an object can be converted to a pointer to `std::byte`, `char` or `unsigned char`
- A pointer can be converted to an integral type (typically `uintptr_t`)
- Invalid conversions usually lead to **undefined behavior**

Strict Aliasing Rule (1)

It is **undefined behavior** to access an object using an expression of different type

- In particular, we are not allowed to access an object through a pointer to another type (pointer aliasing)
- Consequently, compilers typically assume that pointers to different types cannot have the same value
- There are very few exceptions to this rule

Strict Aliasing Rule (2)

foo.cpp

```
static int foo(int* x, double* y) {
    *x = 42;
    *y = 3.0;
    return *x;
}

int main() {
    int a = 0;
    double* y = reinterpret_cast<double*>(&a);
    return foo(&a, y);
}
```

Compiling this with `g++ -O1` will result in the following assembly

foo.o

```
main:
movl    $0, %eax
ret
```

Strict Aliasing Rule (3)

foo.cpp

```
static int foo(int* x, double* y) {
    *x = 42;
    *y = 3.0;
    return *x;
}

int main() {
    int a = 0;
    double* y = reinterpret_cast<double*>(&a);
    return foo(&a, y);
}
```

Compiling this with g++ -O2 will result in the following assembly

foo.o

```
main:
movl    $42, %eax
ret
```

Examining the Object Representation (1)

Important exception to the strict aliasing rule

- Any pointer may legally be converted to a pointer to `char`, or `unsigned char`
- Any pointer may legally be converted to a pointer to `std::byte` (defined in `<cstddef>` header, requires C++17),
- Permits the examination of the *object representation* of any object as an array of bytes

`std::byte` behaves similarly to `unsigned char`

- Represents a raw byte without any integer or character semantics
- Only bitwise operators are defined on bytes

Examining the Object Representation (2)

Example (compile with `g++ -std=c++20`)

```
#include <iostream>
#include <iomanip>
#include <cstddef>

int main() {
    double a = 3.14;
    const std::byte* bytes = reinterpret_cast<const std::byte*>(&a);

    std::cout << "The object representation of 3.14 is 0x";
    std::cout << std::hex << std::setfill('0') << std::setw(2);

    for (unsigned i = 0; i < sizeof(double); ++i)
        std::cout << static_cast<unsigned>(bytes[i]);

    std::cout << std::endl;
}
```

uintptr_t

Any pointer may legally be converted to an integral type

- The integral type must be large enough to hold all values of the pointer
- Usually, `uintptr_t` should be used (defined in `<cstdint>` header)
- Useful in some cases, especially when building custom data structures

Example

```
#include <cstddint>
#include <iostream>

int main() {
    int x = 42;
    uintptr_t addr = reinterpret_cast<uintptr_t>(&x);

    std::cout << "The address of x is " << addr << std::endl;
}
```

The `sizeof` Operator (1)

The `sizeof` operator queries the size of the object representation of a type

```
sizeof( type )
```

Explanation

- The size of a type is given in bytes
- `sizeof(std::byte)`, `sizeof(char)`, and `sizeof(unsigned char)` return `1` by definition
- Depending on the computer architecture, there may be 8 or more bits in one byte (as defined by C++)

The sizeof Operator (2)

The size of an object and pointer arithmetics are closely related

foo.cpp

```cpp
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};

    std::cout << "sizeof(int) = " << sizeof(int) << std::endl;

    int* ptr0 = &array[0];
    int* ptr1 = &array[1];

    uintptr_t uptr0 = reinterpret_cast<uintptr_t>(ptr0);
    uintptr_t uptr1 = reinterpret_cast<uintptr_t>(ptr1);

    std::cout << "(ptr1 - ptr0) = " << (ptr1 - ptr0) << std::endl;
    std::cout << "(uptr1 - uptr0) = " << (uptr1 - uptr0) << std::endl;
}
```

The `sizeof` Operator (3)

On an x86-64 machine, the program might produce the following output

```
$ ./foo
sizeof(int) = 4
(ptr1 - ptr0) = 1
(uptr1 - uptr0) = 4
```

Interpretation

- One `int` occupies 4 bytes
- There is one `int` between `ptr0` and `ptr1`
- There are 4 bytes (i.e. exactly one `int`) between `ptr0` and `ptr1`

The `alignof` Operator

Queries the alignment requirements of a type

```
alignof( type )
```

Explanation

- Depending on the computer architecture, certain types must have addresses aligned to specific byte boundaries
- The `alignof` operator returns the number of bytes between successive addresses where an object of *type* can be allocated
- The alignment requirement of a type is always a power of two
- Important (e.g.) for SIMD instructions, where the programmer must explicitly ensure correct alignment
- Memory accesses with incorrect alignment leads to undefined behavior, e.g. SIGSEGV or SIGBUS (depending on architecture)

Usage Guidelines

When to use references

- Pass-by-reference function call semantics
- When it is guaranteed that the referenced object will always be valid
- When object that should be referenced is always the same

When to use pointers

- Only when absolutely necessary!
- When there may not be a pointed-to object (i.e. `nullptr`)
- When the pointer may change to a different object
- When pointer arithmetic is desired

We will revisit this discussion later during the lecture

- Decision is intricately related to *ownership* semantics
- We would actually like to avoid using raw pointers as much as possible
- There are standard library classes which encapsulate pointers

Troubleshooting

Pointers have a reputation of being highly error-prone

- It is very easy to obtain pointers that point to invalid locations
- Once such a pointer is dereferenced, a number of bad things can happen

Bad things that may happen

- The pointer pointed outside of the program's address space
  - The program will likely segfault immediately
- The pointer pointed outside of the intended memory region, but still inside the program's address space
  - The program might segfault immediately
  - ...or simply corrupt some memory, which might lead to problems later

With the right tools, debugging is not as daunting as it may seem

The Infamous Segfault (1)

Every C++ programmer will encounter a segfault eventually

- Raised by hardware in response to a memory access violation
- In most cases caused by invalid pointers or memory corruption

Obvious example

foo.cpp

```cpp
int main() {
    int* a;
    return *a;  // ERROR: Dereferencing an uninitialized pointer
}
```

Executing this program might result in the following

```
$ ./foo
[1]    5128 segmentation fault (core dumped)  ./foo
```

The Infamous Segfault (2)

Sometimes, the root cause may be (much) more difficult to determine

bar.cpp

```
int main() {
    long* ptr;
    long array[3] = {123, 456, 789};
    ptr = &array[0];
    array[3] = 987;  // ERROR: off-by-one access

    return *ptr;
}
```

When compiled with `g++ -fno-stack-protector`, this will also segfault

- The off-by-one access `array[3] = 987` actually changes the value of `ptr`
- Dereferencing this pointer in the return statement will result in a segfault
- The `-fno-stack-protector` option is required, because `g++` will by default emit extra code to prevent such buffer overflows

The Infamous Segfault (3)

Use the address sanitizer!

```
$ g++ -g -fno-stack-protector -obar bar.cpp
$ ./bar
[1]    4199 segmentation fault (core dumped)  ./bar
$ g++ -g -fno-stack-protector -fsanitize=address -obar bar.cpp
$ ./bar
=================================================================
==4229==ERROR: AddressSanitizer: stack-buffer-overflow on address [...]
WRITE of size 8 at 0x7fff536479d8 thread T0
    #0 0x5617976d529f in main (/tmp/bar+0x129f)
    #1 0x7f6a0fcc3022 in __libc_start_main (/usr/lib/libc.so.6+0x27022)
    #2 0x5617976d50ad in _start (/tmp/bar+0x10ad)

Address 0x7fff536479d8 is located in stack of thread T0 at offset 56 in
↪   frame
    #0 0x5617976d5188 in main (/tmp/bar+0x1188)

    This frame has 1 object(s):
      [32, 56) 'array' (line 3) <== Memory access at offset 56 overflows
↪   this variable
[...]
==4229==ABORTING
```