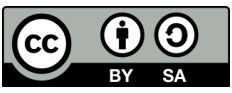
Memory

By Braxton Cuneo



Overview

Course Learning Outcomes
Preface
Data Types
- Primitives
- Arrays
- Structs
- Accessing Arrays and Structs

Storage Types
- Lifetimes
- Static Storage
- Stack Storage
- Dynamic Storage

Pointers and Addresses
- Getting Addresses with
- Identifying Storage with Pointers
- Pointer Type Signatures
- Null Pointers
- Getting Values from Pointers
- Pointer Arithmetic
- Indexing Pointers
- Pointers vs Arrays

Using Dynamic Memory
- Allocating and Deallocating Memory
- Common Memory Bugs
- Common Memory Use Patterns

Course Learning Outcomes

Course Learning Outcomes - Core Concepts

Covered Here

- [x] Pointer variables
- [x] Pointers and arrays
- [x] Pointer arithmetic
- [x] Dynamic memory allocation
- [x] Dynamic arrays

Not Covered Here

- [ ] Pointers to object
- [ ] Constructor
- [ ] Copy Constructor
- [ ] Destructor
- [ ] Overloaded assignment operators (for classes that use dynamic memory)

Course Learning Outcomes - Expectations

Covered Here

- [x] Initialize and manipulate pointers and their values
- [x] Distinguish between execution stack and heap memory allocations
- [x] Understand the relationship between arrays and pointers
- [x] Distinguish between assignment to `nullptr` and `delete`
- [x] Introduce pointer arithmetic operations and comparisons
- [x] Introduce typical cases of memory leaks
- [x] Implement functions that have pointers as parameters and return types

Covered Here - Continued

- [x] Distinguish and appropriately apply the different implementations of `const` (pointers to constants, declaration of a pointer constant, constant pointers, etc.)
- [x] Initialize and manipulate values of 1D and 2D (using array of pointers) dynamic arrays

Not Covered Here

- [ ] Implement constructor, destructor, copy constructor, and overloaded assignment operator for classes that use dynamic memory.
- [ ] Understand and use `this` pointer

Preface

Fixed-Width Integer Types

Integer Types can Vary in Size/Range

TYPE	SIZE	$\geq$	$\leq$
char	1		`short int`
short int	≥ 2		`int`
int	≥ 2	`short int`	`long int`
long int	≥ 4	`int`	`long long int`
long long int	≥ 8	`long int`	

Fixed-Width Integers Cannot

UNSIGNED	SIGNED	SIZE
uint8_t	int8_t	1
uint16_t	int16_t	2
uint32_t	int32_t	4
uint64_t	int64_t	8

***Must #include <cstdint>**

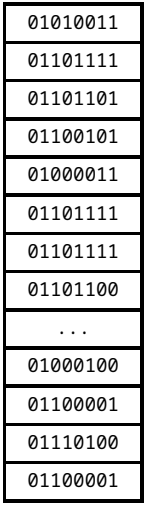
This presentation relies upon diagrams to represent how data can be laid out in memory. To make our type diagrams more consistent across platforms, fixed-width types will be used in these slides.

A note about the diagrams in this slide deck

This slide deck uses a particular format for representing data types and how they are used to assign types and values to the bytes in memory.

The central column of this format is a column of bytes - representing the bytes in memory as they are ordered in memory. The lower a byte is on this column, the higher its address in memory.

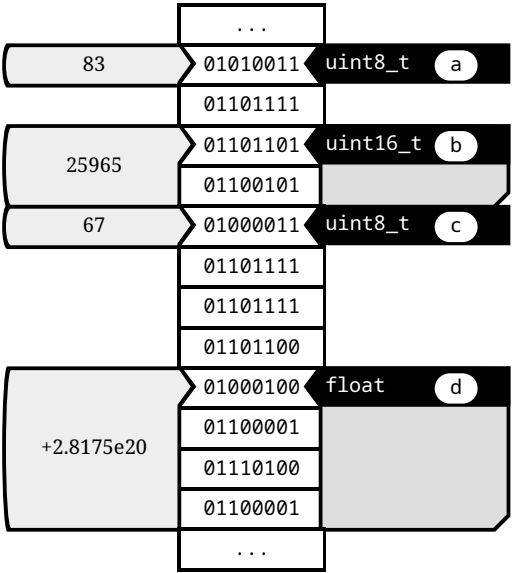
Ellipses indicate series of bytes which are ignored for the purposes of the provided figure. If no ellipses or binary is shown, this indicates that the bytes are unknown or not relevant to the current figure.



A note about the diagrams in this slide deck

The right portion of this format represents the types that are used to describe the displayed bytes. Segments of this right portion which vertically align with the rows in the middle column (the bytes) ascribe meaning to those bytes.

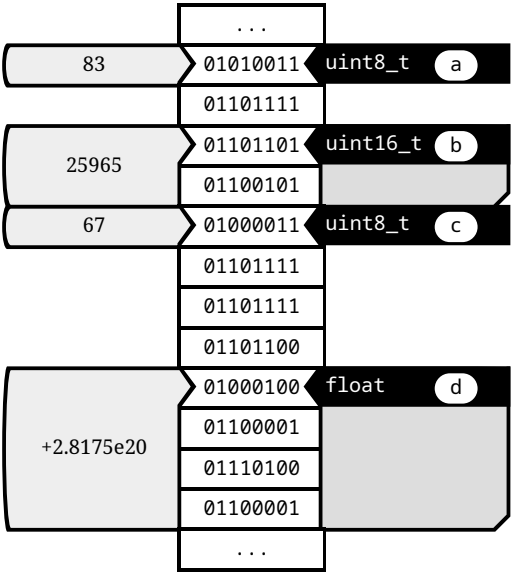
These types may be labeled (in white, pill-shaped regions) with their address, index, or corresponding field name.



A note about the diagrams in this slide deck

The left portion of this format shows the human-readable values that we get from interpreting bytes by their assigned type.

In this fashion, we can see the data in a program as understood by humans (on the left), by the hardware (in the middle), and by the compiler (on the right).



Why are we talking about memory?

- Programs both manipulate and are made of information. Understanding the storage of that information is fundamental to understanding computer science.
- **Even languages like Java and Python are affected by the concepts covered in this lecture.** The standard Python interpreter is written in C, and the openJDK JVM is written in C++. C/C++ is extremely influential.
- Even if you never plan on working with pointers, the alternatives to pointers can behave in very pointer-like ways.

What is Memory?

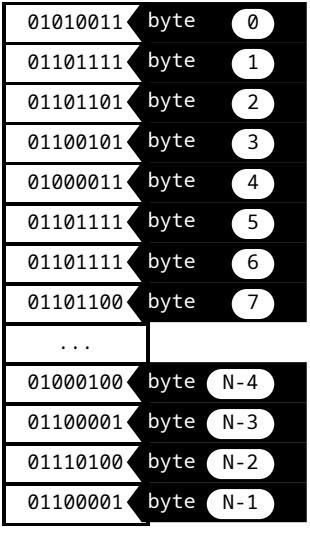
RAM is like one big array that stores all of your variables

A CPU writing/reading RAM is like indexing an array.

If a variable is stored in RAM, the only way that variable can be used is by knowing the corresponding address.

As a compiler translates code into an executable, it plans where variables are going to be stored in memory.

*Not all variables are stored in RAM



Memory in a Nutshell

Things in memory are located in terms of their **base address** – the address of their first byte.

The **type** of a variable allows compilers to determine where that variable's **fields** (elements, members, etc) are relative to its **base address**.

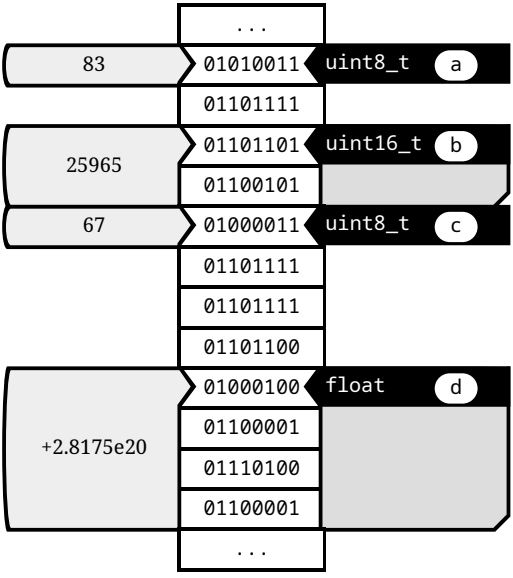
The **storage** of a variable determines what strategy a compiler uses to plan that variable's **base address**.

Data Types

Primitive Types in Memory

Primitive Types

```
1 uint8_t  a;
2 uint16_t b;
3 uint8_t  c;
4 float    d;
```

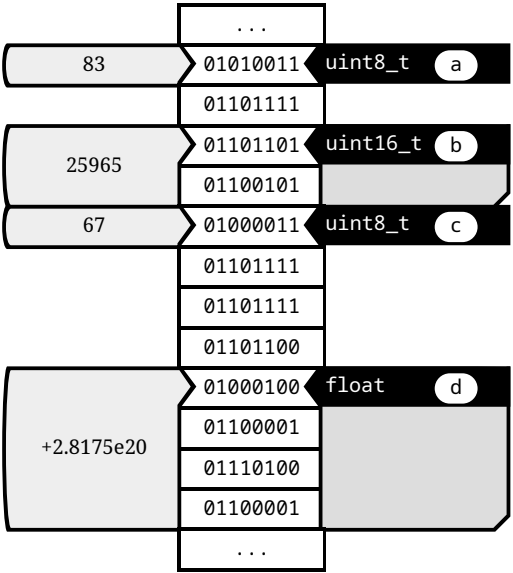


Primitive Types in Memory

Primitive Types

1. The basis of all other data types
2. Have no typed sub-components
3. Always stored as contiguous bytes

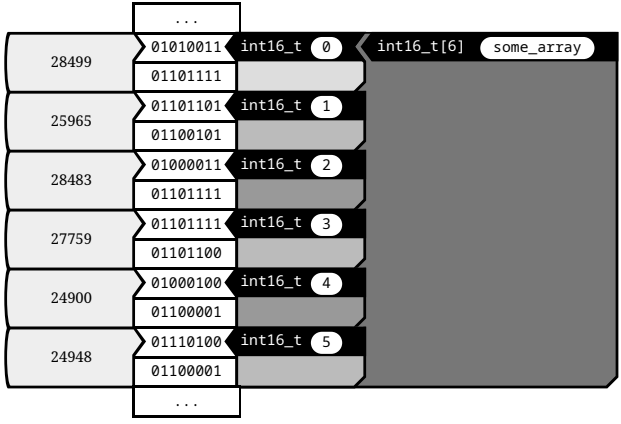
⊕ Aligned by their size (by default)



Arrays in Memory

Arrays

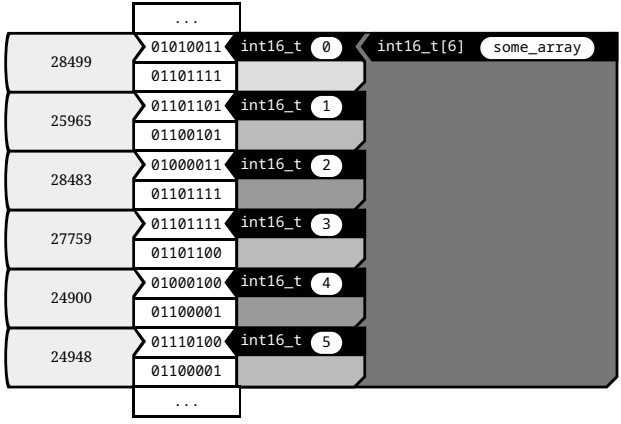
```
1
2 int16_t some_array[6];
3
```



Arrays in Memory

Arrays

1. Store multiples of one type of data
2. Store elements contiguously
3. Store elements by the order of their index

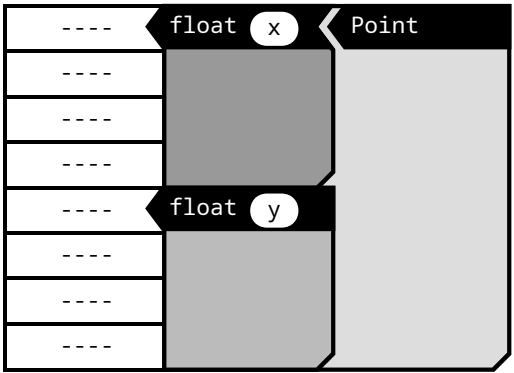


Structs in Memory

Structs in Memory

1. Can store any combination of types
2. Store fields in order declared
3. Store fields contiguously

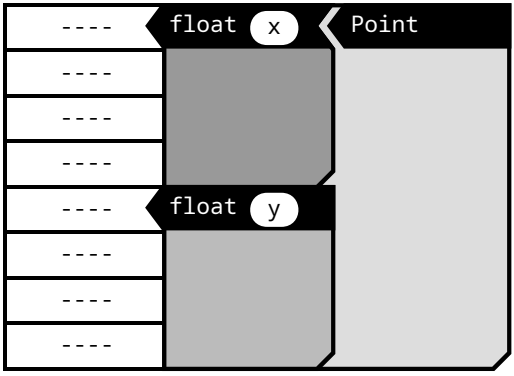
- ⊕ Put padding before fields for field alignment
- ⊕ Put padding at end for struct alignment
- ⊕ Struct alignment = max field alignment



Structs in Memory

Structs in Memory

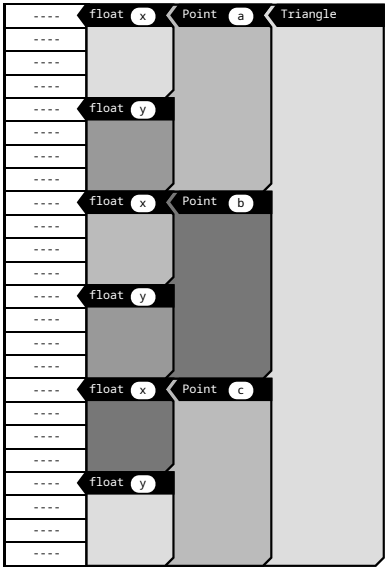
```
1
2 struct Point {
3   float x;
4   float y;
5 };
6
```



Structs in Memory

Structs in Memory

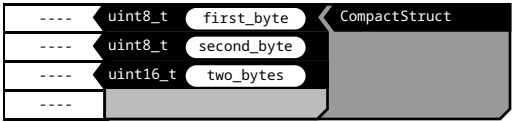
```
1
2  struct Point {
3    float x;
4    float y;
5  };
6
7  struct Triangle {
8    Point a;
9    Point b;
10   Point c;
11 };
12
```



Structs in Memory

Structs in Memory

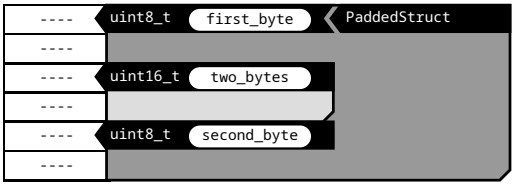
```
1
2 struct CompactStruct {
3   uint8_t  first_byte;
4   uint8_t  second_byte;
5   uint16_t two_bytes;
6 };
7
```



Structs in Memory

Structs in Memory

```
1
2 struct PaddedStruct {
3   uint8_t  first_byte;
4   uint16_t two_bytes;
5   uint8_t  second_byte;
6 };
7
```



Accessing Arrays and Structs

The type of a variable allows compilers to determine where that variable's **fields** (elements, members, etc) are relative to its **base address** - the address of its first byte.

Arrays

To get the element `index` from an array of type `type` starting at address `base_address`, programs perform the following calculation:

`base_address` + `index` × `sizeof(type)`

This is possible because every element in an array is the same type.

Structs

Structs can be made of a mixture of types, but the addresses of their fields cannot be calculated like arrays.

Every time the compiler encounters a struct definition, it develops a plan for where each of its fields are going to go.

Programs evaluate struct-field accesses by using the pre-planned offset of the corresponding field.

Storage Types

Lifetimes

Lifetimes

Things which have value(s) that are stored in memory* are said to have a **lifetime**.

The **lifetime** of something is the span of time when its values are stored in memory.

The lifetime of struct fields and array elements is equal to their containers.

Lifetimes

```
1
2  // Each MyStruct gets its own struct_value
3  struct MyStruct {
4      int struct_value;
5  };
6
7  // Each my_function call gets its own function_value
8  void my_function() {
9      int function_value;
10 }
11
12 // Only one global_value per program.
13 int global_value;
14
15 // Only one global_struct_value per program.
16 MyStruct global_struct_value;
17
```

*These things (arrays, structs, primitives, etc) are collectively referred to as “objects”. This term can be confusing, so these slides will not use it.

Lifetimes

Function Call Lifetimes

While functions do not contain values like structs, it is useful to consider their calls as having "lifetimes".

While a function call is happening, the variables used in that function are stored in memory. The **lifetime** of function call's variables is the period of time when that call is running.

Function Call Lifetimes

```
1 int my_function() {
2   // variable and result only exist while
3   // their function call is happening
4   int variable = 2;
5   int result   = variable * 3;
6   return result;
7 }
```

Types of Storage

Static Storage Stores values that exist for the whole lifetime of a program.

This storage is **automated during compilation**. Every thing in static storage has a **fully pre-determined** offset in program memory.

Stack Storage Stores values that exist for the lifetime of a function call.

This storage is **automated as the program executes**. Every thing in stack storage has a pre-determined offset **defined relative to an address that can move**.

Dynamic Storage Stores values that can be created or destroyed whenever.

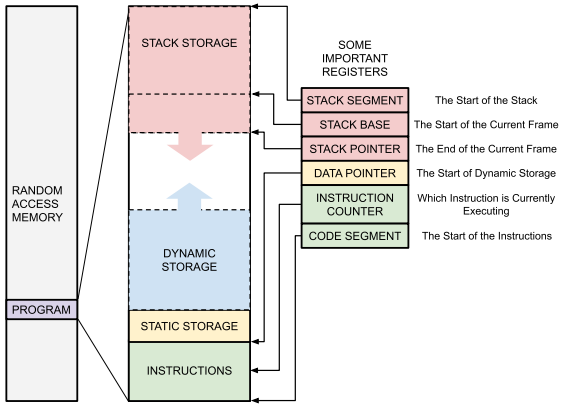
In C/C++, this storage has **no automation** by the compiler. The base addresses of every thing in dynamic memory is **managed by the programmer.**

How Static and Stack Storage Works

Automated storage works by using processor registers to track important offsets in memory.

The offsets tracked for static storage never change during the lifetime of a program.

Programs get the address of a field in static storage by adding its assigned relative offset to the starting address of static storage.

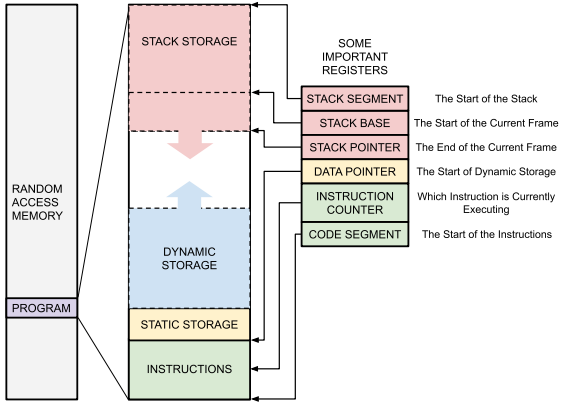


How Static and Stack Storage Works

All function calls store their variables in a region of memory called a **frame**, and each call gets its own frame.

Programs get the address of a variable in the current function call by adding its assigned relative offset to the base address of the current frame.

Programs allocate room for additional function calls by shifting the start and end frame offsets to unused portions of memory. Because of this, **the offsets tracked for stack storage may changed during the lifetime of a program.**



Static Storage

Global (Static) Variables

```cpp
#include <iostream>

// In static storage, and cannot be modified
const uint32_t STATIC_CONSTANT = 2;
// In static storage, and can be modified
uint32_t STATIC_VARIABLE = 0;

void foo() {
    STATIC_VARIABLE += STATIC_CONSTANT;
}

void bar() {
    STATIC_VARIABLE *= STATIC_CONSTANT;
}

int main () {
    foo();
    bar();
    std::cout << "STATIC_VARIABLE: "
              << STATIC_VARIABLE // == 4
              << std::endl;
    return 0;
}
```

Values declared outside of any function, class, struct, etc. are not "tied" to anything that can be created multiple times (eg structs) or called multiple times (eg functions).

All global variables/constants are static.

Not all static variables/constants are in the global scope.

Stack Storage

The variables inside of a function exist only during the lifetime of the corresponding function call.

When a function is not being called, its corresponding variables do not take up memory.

In cases where multiple calls to the same function are occurring simultaneously (eg recursion), each call tracks its own instance of the function's variables in separate regions of memory.

Stack Storage

Stack Storage

```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```

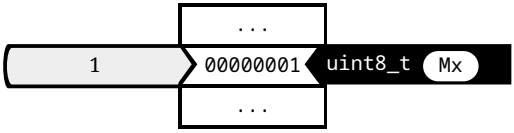
The following is a walk-through of how values are added to and removed from the stack. This walk-through is **simplified**, and some aspects of this walk-through will not be true on every machine, OS, etc.

Highlighted areas show what parts of the code are being evaluated at the time.

Stack Storage

Stack Storage

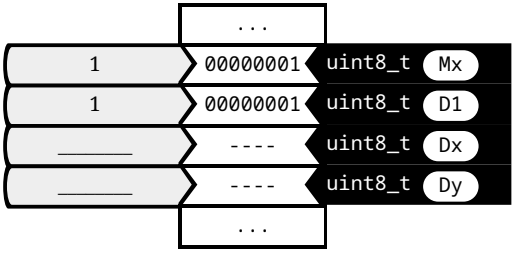
```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```



Stack Storage

Stack Storage

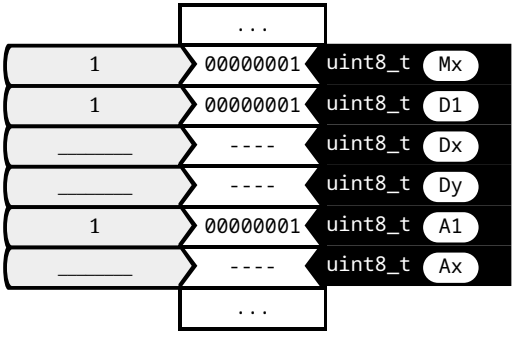
```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```



Stack Storage

Stack Storage

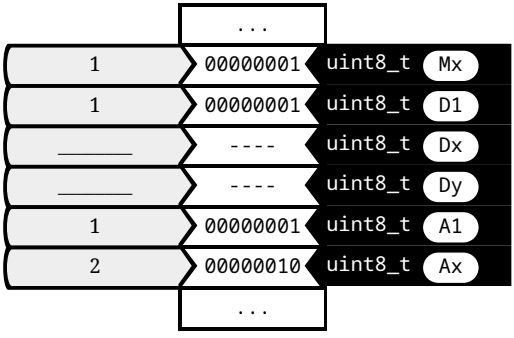
```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```



Stack Storage

Stack Storage

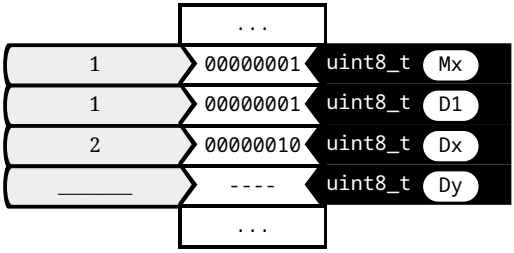
```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```



Stack Storage

Stack Storage

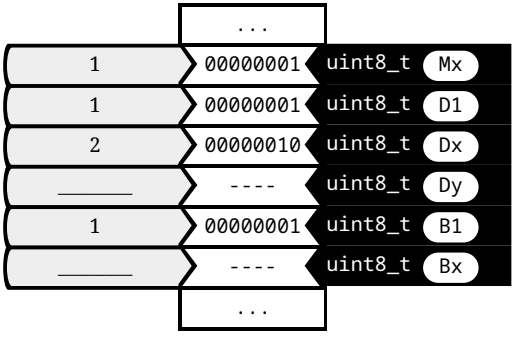
```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```



Stack Storage

Stack Storage

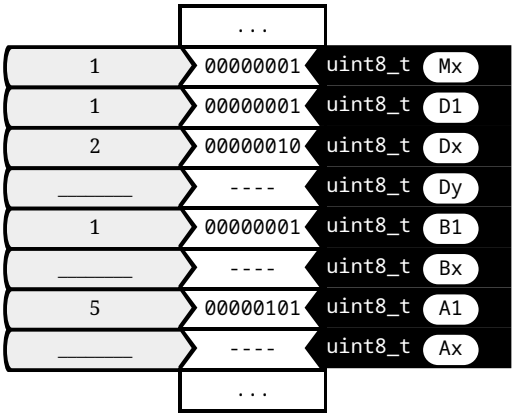
```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```



Stack Storage

Stack Storage

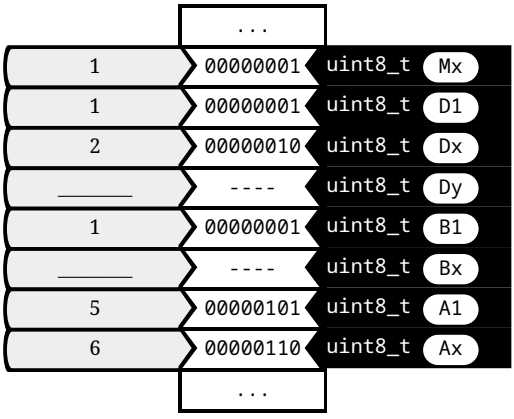
```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```



Stack Storage

Stack Storage

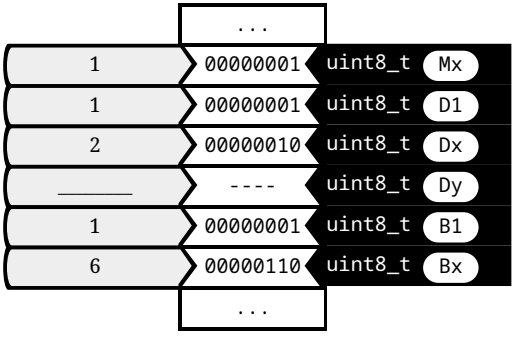
```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```



Stack Storage

Stack Storage

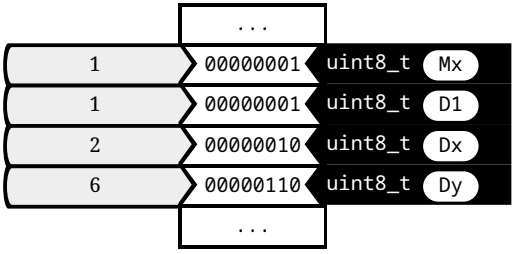
```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```



Stack Storage

Stack Storage

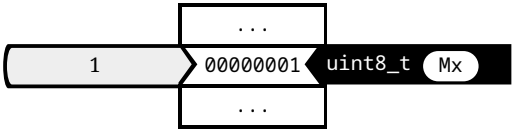
```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```



Stack Storage

Stack Storage

```
1  uint8_t alpha(uint8_t A1) {
2    uint8_t Ax = A1 + 1;
3    return Ax;
4  }
5
6  uint8_t beta(uint8_t B1) {
7    uint8_t Bx = alpha(B1 * 5);
8    return Bx;
9  }
10
11 uint8_t delta(uint8_t D1) {
12   uint8_t  Dx = alpha(D1);
13   uint8_t  Dy = beta (D1);
14   return Dx + Dy;
15 }
16
17 int main () {
18   uint8_t Mx = 1;
19   return delta(Mx);
20 }
```



Dynamic Storage vs Other Types of Storage

Static Storage only works for fields that we know **will only exist once, and for the entire lifetime of our program.**

Stack Storage only works for fields that we know will be **de-allocated in the reverse order of their allocation**. In other words, the fields that have most recently been allocated in stack storage must be the soonest to de-allocate.

Dynamic Storage can store values with lifetimes which may begin or end at any time. **In C/C++, this is not automated.**

Dynamic Storage and the Need for Pointers

- **Remember:** A program can only use a value in storage if its address is known.
- The base addresses of fields in dynamic memory must be explicitly handled in program code.
- To work with addresses, we use a special type of variable called a **pointer**.

Pointers and Addresses

Getting Addresses with '&'

- Some expressions evaluate to something that is stored in memory. These are called **lvalues**.
- By placing the `&` operator to the left of an lvalue, we can get it's address.

Getting Addresses

```
1  struct my_struct {
2    int   a;
3    float b[5];
4  };
5
6  int main () {
7    my_struct array[25];
8
9    std::cout << "The array's base address :"
10             << &array << std::endl;
11
12   std::cout << "The base address of array[7] :"
13             << &array[7] << std::endl;
14
15   std::cout << "The base address of array[7].a :"
16             << &array[7].a << std::endl;
17
18   std::cout << "The base address of "
19             << "array[7].b[3] :"
20             << &array[7].b[3] << std::endl;
21
22   std::cout << "This is a compile-time error"
23             << &( array[7].a + 2 ) << std::endl;
24 }
```

How does the '&' operator work?

- **Remember:** The compiler has a system for organizing the base addresses of values in static and stack storage. So, all global and function variables have a known base address.
- **Remember:** The compiler has a plan for how to access every member of a struct and every element of an array, based off of its base address.
- The `&` operator substitutes an lvalue with the expression used to calculate its address.

Identifying Storage with Pointers

- Pointers store data that identifies storage in memory.
- In C/C++, we can make a pointer identify a specific piece of storage by assigning its address.

⊕ Pointers typically identify storage through an address, but some languages include additional information.

</> Getting Addresses

```
1 // int variable named x
2 int   x   = 12345;
3
4 // a pointer named ptr
5 // that identifies x
6 int *ptr  = &x;
```

Pointer Type Signatures

- All pointers have an associated type, indicating the type of value they identify.
- The type signature of a pointer is the type signature of the identified value followed by an asterisk (‘$*$’).

Pointer Type Signatures

```
1  // an int, float, and int constant
2  int         x  = 12345;
3  float       y  = 9.876;
4  int const   z  = 00000;
5
6  // a pointer to an int
7  int       * xp = &x;
8
9  // a pointer to a float
10 float     * yp = &y;
11
12 // a pointer to an int constant
13 int const   * zp = &z;
```

Pointer Constant Type Signatures

- While the `const` keyword can be placed on the left side of type signatures, it works like the '$*$' symbol when placed on the right.
- If `const` is immediately after the base type, the base type is constant.
- If `const` is immediately after the '$*$', the pointer type is constant.

Const Pointers

```
1  // an int, float, and int constant
2  int         x  = 12345;
3  float       y  = 9.876;
4  const int z  = 00000;
5
6  // a pointer constant to an int
7  int * const         cxp = &x;
8
9  // a pointer constaant to a float
10 float * const       cyp = &y;
11
12 // a pointer constant to an int
13 // constant
14 int const * const czp = &z;
```

Pointers to Pointers

We can use multiple '$*$' in a row to indicate pointers to pointers.

Pointers to Pointers

```
1  // an int, float, and int constant
2  int        x   = 12345;
3  float      y   = 9.876;
4
5  // pointers to the values
6  int    *   xp  = &x;
7  float  *   yp  = &y;
8
9  // pointers to pointers to the values
10 int    **  xpp = &xp;
11 float  **  ypp = &xp;
```

Pointers and Functions

One can have pointers as return types or parameters by including the corresponding type signature.

</> Pointers and Functions

```
1  int const MAGIC[2] = {0,1};
2
3  int const *magic_value(bool choice){
4    if( choice ){
5      return &MAGIC[0];
6    }
7    return &MAGIC[1];
8  }
9
10 void print_magic(int const * pointer){
11   std::cout << pointer << std::endl;
12 }
```

Null Pointers

- Pointers do not have to identify valid storage.
- C++ has a special address called `NULL`, which is used to indicate that a pointer does not identify valid storage.
- To make a pointer `NULL`, you assign with `nullptr`.

Null Pointers

```
1  int a = 3;
2  int b = 4;
3
4  // I don't know if ptr will be set
5  // to identify a or b. Until its
6  // value is determined, ptr
7  // shoud be NULL.
8  int *ptr = nullptr;
9
10 if ( condition ) {
11   ptr = &a;
12 } else {
13   ptr = &b;
14 }
```

Getting Values from Pointers

- To access the storage identified by a pointer, you use the dereferencing (‘$*$’) operator.
- Much like how the ‘&’ operater turns an lvalue into an address, the ‘$*$’ operator turns an address into an lvalue.

Dereferencing

```
1 int a = 1;
2
3 int *a_ptr = &a;
4
5 *a_ptr = 2;
6
7 // Will print 2
8 std::cout << a << std::endl;
```

Getting Values from Pointers

If a pointer is dereferenced when it is `NULL`, or when it is pointing to invalid storage **undefined behavior occurs**.

This usually means that the program crashes (eg a segfault), or wrong values will be accessed.

Bad Dereferencing

```
1
2  int a = 1;
3  int *a_ptr;
4
5  // BAD: Pointer uninitialized
6  std::cout << *a_ptr << std::endl;
7
8  a_ptr = nullptr;
9
10 // BAD: Pointer is NULL
11 std::cout << *a_ptr << std::endl;
12
```

Pointer Arithmetic

Given two pointers `P` and `Q`, of type `type*`, and an integer `N`, we can perform the following arithmetic operations.

Operation	Description
`P + N`	Adds `N` * `sizeof(type)` to the address of `P`
`P - N`	Subtracts `N` * `sizeof(type)` from the address of `P`
`P += N`	Works like `P + N`, with assignment to the pointer
`P -= N`	Works like `P - N`, with assignment to the pointer
`P ++`	Works like `P += 1`
`P --`	Works like `P -= 1`
`P - Q`	Returns the difference in addresses, divided by `sizeof(type)`
`P == Q`	Returns `true` only for identical addresses
`P != Q`	Returns `false` only for identical addresses

Pointer Arithmetic

Pointer/Integer Arithmetic

```cpp
char alpha[8] = {'A','B','C','D','E','F','G','H'};

char *p = &alpha[0];
char *q = &alpha[6];

std::cout << *p << std::endl; // -> 'A'
std::cout << *q << std::endl; // -> 'G'

std::cout << *(p+3) << std::endl; // -> 'D'
std::cout << *(q-2) << std::endl; // -> 'E'

p += 3;
q -= 2;

std::cout << *p << std::endl; // -> 'D'
std::cout << *q << std::endl; // -> 'E'

p--;
q++;

std::cout << *p << std::endl; // -> 'C'
std::cout << *q << std::endl; // -> 'F'
```

Pointer/Pointer Arithmetic

```cpp
char alpha[8] = {'A','B','C','D','E','F','G','H'};

char *p = &alpha[0];
char *q = &alpha[6];

std::cout << *p << std::endl; // -> 'A'
std::cout << *q << std::endl; // -> 'G'

std::cout << (p-q) << std::endl; // -> 6

p += 2;
std::cout << *p << std::endl; // -> 'C'
std::cout << *q << std::endl; // -> 'G'

std::cout << (p == q) << std::endl; // -> 1 (false)
std::cout << (p != q) << std::endl; // -> 0 (true)

q -= 4;

std::cout << (p-q)    << std::endl; // -> 0
std::cout << (p == q) << std::endl; // -> 1 (true)
std::cout << (p != q) << std::endl; // -> 0 (false)
```

Indexing Pointers

Indexing a pointer is equivalent to dereferencing the sum of the pointer and the index.

In other words, given a pointer `P` and an integer `N`:

- `P[N]` is equivalent to `*(P+N)`
- `P[0]` is equivalent to `*P`
- `N[P]` is equivalent to `P[N]`

Indexing Pointers

```
1  char alpha[4] = {'A','B','C','D'};
2
3  char *p = &alpha[1];
4
5  std::cout <<  *p    << std::endl; // -> 'B'
6  std::cout <<   p[0] << std::endl; // -> 'B'
7
8  std::cout << *(p+2) << std::endl; // -> 'D'
9  std::cout <<   p[2] << std::endl; // -> 'D'
10 std::cout <<   2[p] << std::endl; // -> 'D'
11
12 std::cout << *(p-1)   << std::endl; // -> 'A'
13 std::cout <<  p[-1]   << std::endl; // -> 'A'
14 std::cout <<  (-1)[p] << std::endl; // -> 'A'
```

Pointers vs Arrays

Arrays are automatically cast into pointers to their first element when they are:

- indexed
- assigned to a pointer variable
- passed as a pointer parameter
- returned from a function that returns a pointer

However, arrays and pointers are different types of variables.

Arrays store a contiguous block of elements in memory. Pointers simply identify storage.

Pointers vs Arrays

```
1
2  uint32_t array[10] =
   ↪  {0,1,2,3,4,5,6,7,8,9};
3
4  uint32_t *pointer = array;
5
6  // An array stores its values
7  std::cout << sizeof(array)
8            << std::endl; // -> 40
9
10 // A pointer identifies storage
11 std::cout << sizeof(pointer)
12           << std::endl; // ->  8
13
```

Using Dynamic Memory

Allocating Memory

In C++, dynamic storage may be allocated through the `new` keyword.

Given some type `type`:

- `new type` returns the address of a newly allocated `type` in dynamic memory
- `new type[N]` returns the address of a newly allocated array with `N` `type` elements in dynamic memory

Allocating Memory

```
1
2 // Single-value allocation
3 int *single_value = new int;
4
5
6 // Array allocation
7 int *array = new int[10];
8
9
```

Deallocating Memory

In C++, dynamic storage may be deallocated through the `delete` keyword.

Given a pointer `P` pointing to dynamically allocated memory:

- `delete P` deallocates the value identified by `P`
- `delete[] P` deallocates the array identified by `P`

Deallocating Memory

```
1 int *single_value = new int;
2 int *array = new int[10];
3
4 // Single-value deallocation
5 delete single_value;
6
7 // Array deallocation
8 delete[] array;
```

`delete` should only be used for single-value allocations and `delete[]` should only be used for array allocations.

Common Dynamic Memory Bugs

1. Mixing up `delete` and `delete[]`
2. Using storage before initialization
3. Using storage after deallocation
4. Double-freeing storage
5. Losing dynamic memory base addresses before they are deallocated
6. Returning reference/pointers to expiring stack storage
7. Reading and writing out of array bounds

Mixing up `delete` and `delete[]`

There is a difference between single-value and array allocations, and they must be deallocated differently.

`delete` should only be used for single-value allocations and `delete[]` should only be used for array allocations.

If you use the wrong keyword, the information used to track memory allocations will get messed up.

Mixing up Deletions

```
1 int *single_value = new int;
2 int *array = new int[10];
3
4 // Do not do this
5 delete[] single_value;
6
7 // Do not do this either
8 delete   array;
```

Using Storage Before Initialization

You should not:

- Use variables before they are initialized
- Dereference pointers before they are given a valid address
- Use the storage identified by pointers before it is initialized

Junk Data

```
1  int  value;
2  int *pointer;
3  int *dyn_value = new int;
4
5  // Junk data
6  std::cout<<value << std::endl;
7
8  // Junk data or segfault
9  std::cout<<*pointer << std::endl;
10
11 // Junk data
12 std::cout<<*dyn_value << std::endl;
```

Using Storage After Deallocation

If storage has already been freed, it may be used by a subsequent allocation.

Using storage after freeing it can mess up the information used to track memory allocations, or cause a segfault.

Use After Free

```
1 int *pointer = new int;
2
3 delete pointer;
4
5 // Who knows what stuff this could
6 // be overwritting...
7 pointer = 12345;
8
```

Double-Freeing Storage

If storage has already been freed, it may be used by a subsequent allocation.

Freeing the same pointer twice in a row without an allocation in between can mess up the information used to track memory allocations, or cause a segfault.

Double-free

```
1 int *pointer = new int;
2
3 delete pointer;
4
5 // Not good...
6 delete pointer;
7
```

Losing dynamic memory base addresses before deallocation

In order to delete dynamic storage, a pointer to that storage must be supplied.

If we lose the information needed to supply this pointer, we can never free the corresponding storage.

This is called a memory leak.

Losing Pointers

```
1  int calculate(int value) {
2
3    // This calculation requires a
4    // variable amount of storage
5    int   size = needed_size(value);
6    int *array = new int[size];
7
8    /* ... do some processing ... */
9
10   return result;
11
12   // Wait! We forgot to use
13   // delete[] on our array!
14 }
```

Pointers to Stack Storage

Remember: The stack storage used by a function exists only for the lifetime of the corresponding function call.

If we return a pointer identifying storage used by the current function, that storage will be invalid by the time we reach the function we return to.

Expired Storage

```
1  int *dangerous() {
2    int some_value = 1234;
3
4    return &some_value;
5  }
6
7  int main() {
8
9    // Junk data or segfault
10   std::cout << *dangerous()
11             << std::endl;
12
13 }
```

Reading and Writing out of Array Bounds

Remember: Array indexing is just pointer addition and a dereference.

If you use an index that goes past either end of an array, you may access unrelated values or cause a segfault.

Out of Bounds

```
1  int main() {
2
3    int array[4] = { 1,2,3,4 };
4
5    // Junk data or segfaut
6    std::cout << array[-1]
7              << std::endl;
8
9    // Junk data or segfaut
10   std::cout << array[4]
11             << std::endl;
12
13 }
```

Common Dynamic Memory Patterns

1. References
2. Dynamically sized arrays
3. 2D Dynamic Arrays
4. Optional Values

References

References are pointers that:

- must be bound to a variable during declaration
- cannot change what storage they identify
- always automatically dereference to their identified storage

In many ways, references are much safer than pointers, but they can't do as many things.

Additionally, references can still point to storage that can be later invalidated (eg, references to stack storage).

</> Pointers vs References

```
1  int main() {
2
3    int value = 1234;
4
5    // These are equivalent
6    int *ptr = &value;
7    int &ref =  value;
8
9    // These are equivalent
10   std::cout << *ptr << std::endl;
11   std::cout <<  ref << std::endl;
12
13 }
```

Dynamically-Sized Arrays

With dynamic array allocation, we can create types that can store different amounts of information.

However, to use dynamic arrays safely, we must be sure to remember the size of the array we are using, so we do not write out of bounds.

Dynamic Arrays

```
1
2  struct dyn_array {
3    int  size;
4    int *array;
5  };
6
7  dyn_array make_array (int size) {
8    dyn_array result;
9    result.size  = size;
10   result.array = new int[size];
11 }
12
```

2D Dynamic Arrays

We can dynamically allocate storage for pointers, just like other types.

By allocating an array of pointers and initializing each of those pointers with a dynamically allocated array, one can use the pointer to the array-of-pointers like a normal 2D array.

However, you must remember to dynamically allocate each sub-array before you use it and deallocate the sub-arrays before you deallocate the array-of-pointers.

2D Dynamic Arrays

```
1
2  int **square_array_maker(int size){
3
4    // Allocate our array of pointers
5    int **result = new int*[size];
6
7    // Allocate each sub-array
8    for(int i=0; i<size; i++) {
9      result[i] = new int[size];
10   }
11
12   return result;
13
14 }
15
```

Optional Values

Remember: Dynamic memory lets us represent any number of values, including zero values.

If we want a field to be optional, we can represent it as a pointer, with `NULL` pointers indicating that no such value exists.

This is particularly valuable if the optional value is very large, because that means we only use the extra memory if we need it.

Optional Values

```
1
2  struct fancy_data {
3    int first_value;
4    int *second_value;
5  }
6
7  void fancy_print(fancy_data& data) {
8    std::cout << "First value: "
9              << data.first_value
10             << std::endl;
11    if ( data.second_value == nullptr ) {
12      std::cout << "There is no second value"
13                << std::endl;
14    } else {
15      std::cout << "Second value: "
16                << *data.second_value
17                <<std::endl;
18    }
19 }
20
```