C/C++ PRIMER

LECTURE 2: BUILT-IN TYPES, ARRAYS AND POINTERS

Fabian Wermelinger
Harvard University

OUTLINE

- Built-in types
- Arrays
- Pointers
- Difference between pointers and arrays

BUILT-IN TYPES

EXPRESSION STATEMENTS

Expression statements are the most common statements in typical C/C++ code. They must be terminated with a semi-colon " ; ".

EXAMPLE: EXPRESSION STATEMENTS

```
1 // An expression followed by a semicolon is a statement
2 // Expression can be:
3 //   Variable definition: [storage modifier] type variable_name [assigned value
4 //   Arithmetic expression
5 int a = 1;
6 const float b = 2.1;
7 a = a + b;
```

What happens in line 7?

BASIC BUILT-IN TYPES

There are only few basic types built-in the language

- Integral types (integer)
- Floating-point types (real numbers)
- Character types (ASCII characters)
- Boolean types (true or false)
- Void type (incomplete type)

INTEGER TYPES

Integers are fundamental types in any program. They are usually 32-bit or 64-bit in size and can be *signed* or *unsigned*.

EXAMPLE: INTEGRAL TYPES

```
1 int a = 1;           // signed integer, 32-bit
2 signed int a = 1;    // same as above
3 unsigned int a = 1;  // no negative values! (undefined if negative)
4 short int a = 1;     // optimized for space, at least 16-bit
5 long int a = 1;      // at least 32-bit, same as int a = 1; on most systems
6 long long int a = 1; // at least 64-bit, since C++11
```

INTEGER TYPES

EXAMPLE: INTEGER OVERFLOW

In the following example, what is the value assigned to `a`?

```
1 unsigned int a = -1;
```

```
1 a = 4294967295
```

The compiler sets *all* bits in `a` to `1`, the unsigned type overflows.

We can replicate this result in `python`:

```
1 >>> a = 0xffffffff  # hex notation: all 32 bits set to 1
2 >>> a
3 4294967295
```

Be careful when working with unsigned integral types and you perform operations that can result in negative numbers, for example subtraction. These are difficult bugs to spot!

FLOATING-POINT TYPES

There are two different sizes for representing floating point numbers (numbers in $\mathbb{R}$):

- *Single precision*: 32-bit size, `float` type
- *Double precision*: 64-bit size, `double` type

```
1 float single_precision = 1.0f;  // 32-bit size, IEEE 754 standard
2 double double_precision = 1.0;  // 64-bit size, IEEE 754 standard
```

CHARACTER TYPES

Characters are represented by the `char` type. They are usually 8-bits (= 1 byte), which is enough for ASCII characters.

```
1 char c = 'a';  // assign character 'a' to variable c
2 char d = 97;   // assign integral ASCII code: 97 = 'a' (see ASCII table)
```

BOOLEAN TYPES

Boolean types can take two values (binary). Examples are `0` or `1`, `false` or `true`. Historically, this was achieved with an integral type. The type `bool` exists from the C99 standard onwards and in C++.

```
1 bool yes = true;  // value other than 0
2 bool no = false;  // value 0
```

VOID TYPE

The `void` type is special. It is an empty type and can not be assigned any value. It makes sense for pointers (`void` can point to anything) or function return values if the function does not return anything (more later).

```
1 void my_function();  // function declaration that returns nothing
2 void variable;       // is this expression correct?
```

```
1 $ g++ void.cpp
2 void.cpp: In function ‘int main()’:
3 void.cpp:3:10: error: variable or field ‘variable’ declared void
4     3 |     void variable;
5       |          ^~~~~~~~
```

You can not declare a variable with `void` type!

TYPE QUALIFIERS

You can add type qualifiers to your declarations. There are mainly two type qualifiers:

- `const` : the variable is declared as constant (you will not be able to modify its value)
- `volatile` : prevents the compiler from doing certain optimizations (read/write reordering). Useful for embedded programming and signal handlers (you will encounter it rarely).

```
1 const int a = 1;  // a is declared constant
2 int const a = 1;  // same as above
3 a = 2;            // compile-time error: assignment of read-only variable ‘a’
```

COMPOUND TYPES

There are a number of *compound* types which are a combination of built-in types (or user defined classes in C++, more later). The most common are:

- Structures
- Enumerations
- Arrays

COMPOUND TYPES

EXAMPLE: STRUCTURES

```
1 struct MyStructure {
2     int count;    // the number of 'things' I want to model
3     double value; // the value of the 'things'
4 };
5
6 MyStructure s = {0}; // initialize all bits to 0
7 s.count = 10;        // assign a count of 10
8 s.value = 100.0;     // assign a value of 100
```

COMPOUND TYPES

EXAMPLE: ENUMERATIONS

```
 1 // see: https://en.cppreference.com/w/cpp/language/enum
 2 enum Color { RED, GREEN, BLUE }; // unscoped enum; C-style
 3 void what_color(const Color c)
 4 {
 5     switch (c) {
 6         case RED:   std::cout << "Red\n"; break;
 7         case GREEN: std::cout << "Green\n"; break;
 8         case BLUE:  std::cout << "Blue\n"; break;
 9     }
10 }
11 int c = RED; // implicit cast to integer OK!
12
13 enum class Color { RED, GREEN, BLUE }; // scoped enum; C++, note the 'class' k
14 void what_color(const Color c)
15 {
16     switch (c) {
17         case Color::RED:   std::cout << "Red\n"; break;
18         case Color::GREEN: std::cout << "Green\n"; break;
19         case Color::BLUE:  std::cout << "Blue\n"; break;
20     }
21 }
22 int c = Color::RED; // implicit cast to integer NOT OK!
```

ARRAYS

Arrays are a sequence of elements with the *same type* contiguously laid out in memory. The number of those elements does not change during the array lifetime.

- By default, arrays are allocated on the stack. If you choose too many elements in your array, you may run into a *stack overflow*.
- The number of elements in the array must be known at compile-time

EXAMPLE: ARRAY DECLARATIONS

```
1 int array[10] = {0};  // array of 10 signed integers, initialized to 0
2
3 #define N 10  // can use the pre-processor to replace all occurrences of N with
4 int array[N] = {0};  // same as above
5
6 int array[] = {0, 1, 2, 3};  // initialize with values (compiler knows the size
7 int array[1000000000] = {0}; // compiles but runtime error: segmentation fault!
```

ARRAYS

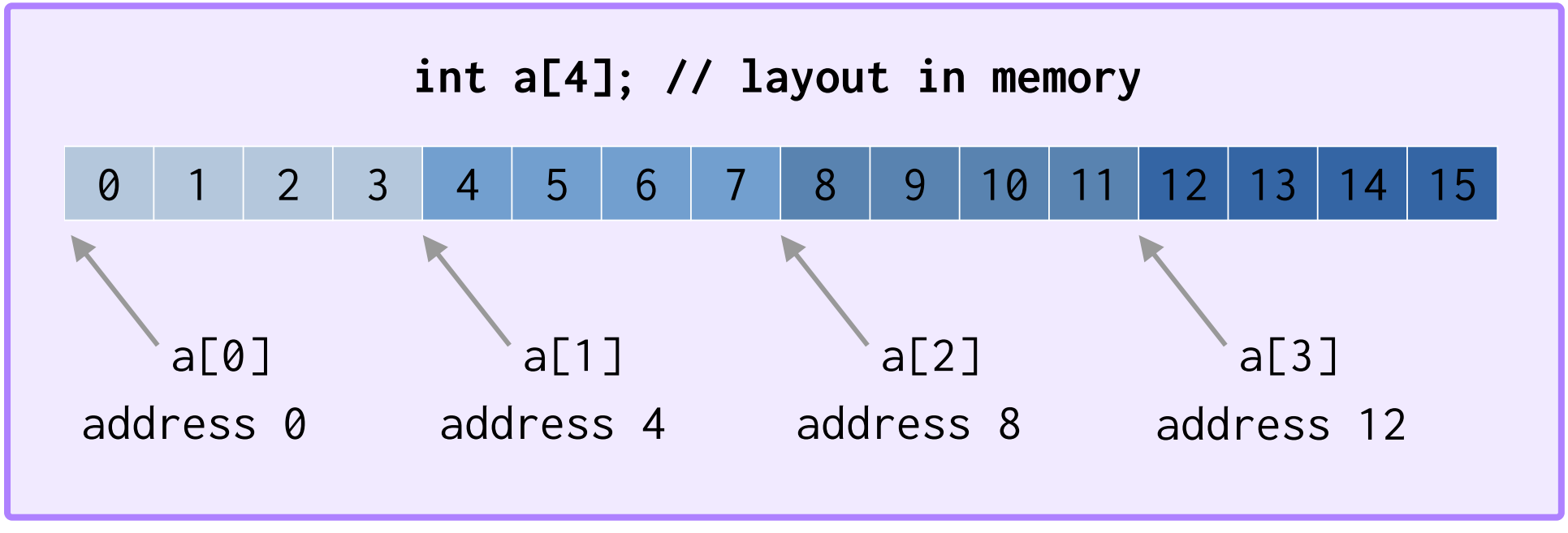
EXAMPLE: ARRAY DECLARATIONS (CONT.)

```cpp
#include<array>
std::array array_cpp = {0, 1, 2, 3}; // C++ way of declaring an array

int array_c[4] = {0, 1, 2, 3};
void my_function(int a[]); // you can pass arrays a function arguments

my_function(array_c);   // OK
my_function(array_cpp); // NOT OK! array_c and array_cpp have different types!

```

Hands-On: 20 minutes (`hands-on/01/README.md`)
Work through the tasks in the `README.md` file, proceed step-by-step.

POINTERS

Memory layout of an array: assume each tile corresponds to a byte and memory addresses start at 0.



In this example, the `int` type is 32-bit and therefore each `int` requires *4 byte* of storage.

POINTERS

In general, a *pointer* is a type of a variable and it only stores a *reference* to a particular variable of that type. Such a reference translates to a *memory address*.

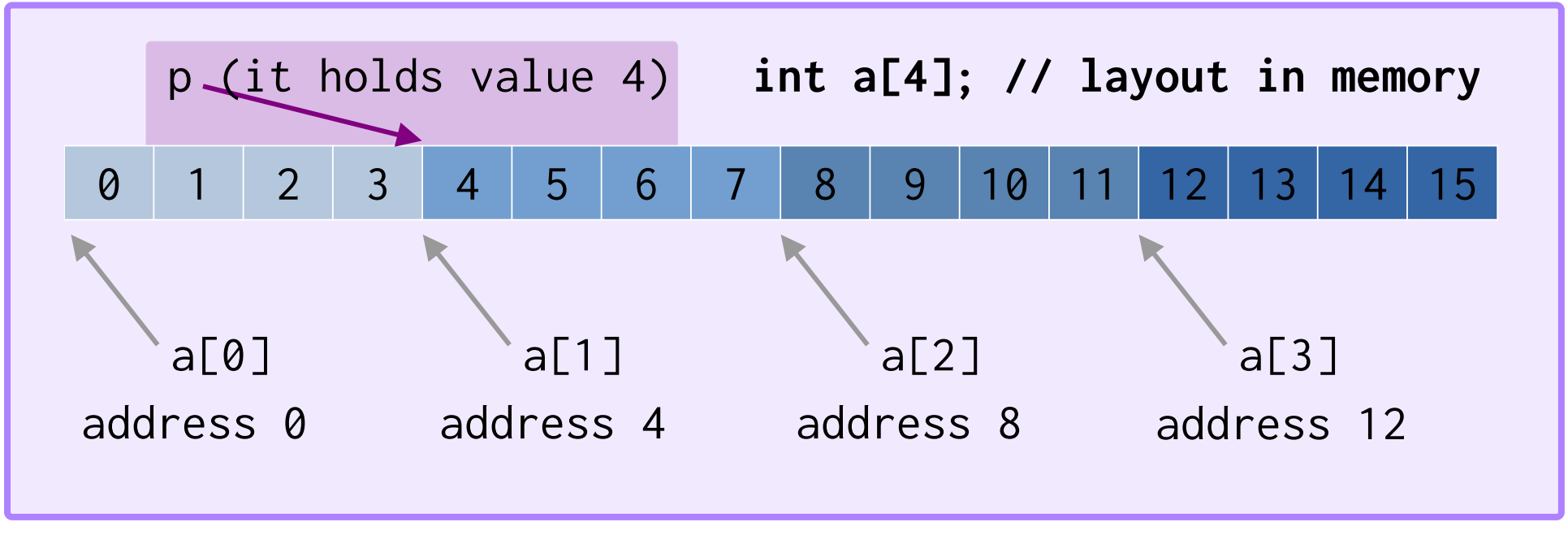
```
1 int a[4];  // array a of integer type
2 int *p;    // pointer of integer type (not initialized)
3
4 p = &a[1]; // assign pointer p the address of the second element in a
5 *p = 2;    // set the value of a[1] to 2 through the pointer
6 p = 2;     // set the pointer address to 2. This will cause mayhem!
```

- `&` is called the address-of operator (or *reference* operator)
- A pointer is declared using an asterisk "`*`"
- A pointer is *dereferenced* using the `*` operator (or *dereference* operator)
- The main value a pointer holds is a *memory address*. To assign a meaningful value to that memory location, you must *dereference* the pointer! (line 5 in the code above)

POINTERS

You can assign any memory address to a pointer. Memory addresses increment by *one byte* at a time.

```
1 int a[4];
2 int *p;
3 p = &a[1]; // this assigns p the memory address 4
4 *p = 2;    // this assigns the value 2 at memory address 4 (same as a[1] = 2)
```



POINTERS

We could use a pointer in the same way as we use an array:

```
1 int a[4];
2 int *p = &a[0]; // you could also write 'int *p = &a;' but being explicit is pr
3 p[1] = 2;       // this assigns the value 2 at memory address 4 (same as a[1] =
```

POINTER ARITHMETIC

- You can compute a relative offset from the current memory address a pointer holds.
- The actual offset depends on the type of the pointer.
- In line 2 above: `p` holds the memory address 0. Because an `int` is *4 bytes*, the expression `p + 1` would produce memory address **4**. The size of one `int` element is 4 bytes.
- The expression `p[1]` is a short hand for `*(p + 1)`. Instead of `p[1] = 2` you could also write `*(p + 1) = 2`.

POINTERS

THE VOID POINTER

As a pointer holds a memory address, it is perfectly legal for a pointer to have a `void` type!

```
1 int a[4];
2 void *p = &a[0]; // this is OK, the pointer p only holds memory address 0
3 p[1] = 2;        // this is no longer OK!!
4 *p = 0;          // this is also not OK!!
```

But, you can not dereference a `void` pointer! (Same as you can not define a variable of `void` type.)

POINTERS

THE VOID POINTER

Where is this useful? Everywhere where you only care about a memory address, but not the particular data that is stored at this address.

```
1 // void* malloc(size_t bytes): allocates bytes on the heap and returns a void p
2 //                             To actually use the memory, we must cast it to i
3 // see: https://en.cppreference.com/w/cpp/memory/c/malloc
4 int *a = (int*)malloc(4 * sizeof(int)); // allocate 4 integers on the heap
```

We will look at memory allocation in a later lecture.

POINTERS VS. ARRAYS

Pointers and arrays are very similar, but they are not the same!

Pointer	**Array**
Holds the *address* of data	Holds data
Data is accessed *indirectly* through the *dereference operator* (" `*` ")	Data is accessed *directly* using `a[i]` statements
Commonly used for *dynamic* data structures	Commonly used for *static* data (you can not change the number of elements)
Typically points to *anonymous* data (this can be dangerous if you don't know what you're doing). You can change to address a pointer points to as you like.	Is a named variable in its own right. You can not assign another array to it after you declared it.

POINTERS VS. ARRAYS

EXAMPLE: ARRAY FUNCTION ARGUMENTS DECAY TO POINTERS

```c
// function with an array argument
extern void my_function(int a[]);
int main(void)
{
    int ary[4];
    int *ptr = &ary[0];
    my_function(ary); // OK, array argument decays to a pointer when compiled
    my_function(ptr); // therefore, passing a pointer is valid (but pointers a
                      // arrays have different types!)
    return 0;
}
```

POINTERS VS. ARRAYS

EXAMPLE: ARRAY FUNCTION ARGUMENTS DECAY TO POINTERS

```c
// function with an array argument
// extern void my_function(int a[]);
extern void my_function(int *a); // this will also compile fine!
int main(void)
{
    int ary[4];
    int *ptr = &ary[0];
    my_function(ary); // OK, array argument decays to a pointer when compiled
    my_function(ptr); // therefore, passing a pointer is valid (but pointers a
                      // arrays have different types!)
    return 0;
}
```

POINTERS VS. ARRAYS

Handling strings with pointers and arrays is different:

```
1 int main(void)
2 {
3     char string_array[] = "Hello world!";    // assign string literal (array)
4     char *string_ptr = "Hello world!";       // assign string literal (pointer)
5     string_array[6] = 'W'; // OK
6     // string_ptr[6] = 'W'; // forbidden! read-only
7     return 0;
8 }
```

compiling this code generates a warning:

```
1 strings.cpp: In function ‘int main()’:
2 strings.cpp:4:24: warning: ISO C++ forbids converting a string constant to ‘cha
3     4 |     char *string_ptr = "Hello world!";
4       |                        ^~~~~~~~~~~~~~
```

POINTERS VS. ARRAYS

Handling strings with pointers and arrays is different:

```
1 int main(void)
2 {
3     char string_array[] = "Hello world!";    // assign string literal (array)
4     const char *string_ptr = "Hello world!"; // assign const string literal (po
5     string_array[6] = 'W'; // OK
6     // string_ptr[6] = 'W'; // forbidden! read-only
7     return 0;
8 }
```

To fix this warning the string literal assigned to a pointer **must** be `const` . This explains the commented line 6 above!

POINTERS AND CONST TYPE QUALIFIERS

Let's revisit the `const` type qualifier for pointers:

- Pointers have a dual characteristic: they hold a memory address which could be changed **and** they can dereference data which could be changed!
- The `const` type qualifier can be used for both of these characteristics of pointers

```
1 int a;
2 int *p;              // can change the address p holds and can change the data p
3 const int *p;        // can change the address p holds but not the data it point
4 int * const p = &a;       // can not change the address p holds but the data it
5 const int * const p = &a; // both are const and can not be changed
```

Note that in the latter two cases, you **must** initialize the pointer when you declare it because the value it holds (the memory address) is constant.

LAST NOTE ABOUT POINTERS

Conceptually pointers are easy. Practically they are one of the hardest language features to apply correctly. They are very important and you have to be proficient with them.

- https://en.cppreference.com/book/pointers
- https://www.cplusplus.com/doc/tutorial/pointers/

Hands-On: Remainder (`hands-on/02/README.md`)
Work through the pointer tasks in the `README.md` file.