Outline

- In this lesson, we will:
  - Revisit the concept of an address
  - Consider how to store them
  - Look at some examples and naming conventions
  - Understand the size of addresses

Addresses

- Recall that we have seen that
  - Memory is byte addressable
    - Every byte has a separate address
    - It is convenient to represent these addresses using hexadecimal
  - The size of an address is fixed for a specific processor
    - Most general processors have 64-bits addresses
    - Microcontrollers generally have up to 32-bit addresses
    - The PIC 10F200 microcontroller has 8-bit addresses
      - It costs less than 50¢

Addresses

- The different integer types store various numbers of bits:

Type	Bits	
`unsigned char`	8	PIC 10F200
`unsigned short`	16	
`unsigned int`	32	Intel 486
`unsigned long`	64	Most computers today…

  - Question: Can we not just store an address?

Addresses

- We could have a primitive data type 'address_t':

```
int main() {
    int array[10];
    address_t addr_array{ array };

    // The variable 'addr_array' is now stores
    // the address of 'array'

    // These should print the same value
    std::cout << array << std::endl;
    std::cout << addr_array << std::endl;
    // Carry on...

    return 0;
}
```


Addresses

- Problem: We have an address, but what is there?
  - Is it an int?
  - Is it a double or a float?

```
int main() {
    int array_a[10];
    double array_b[10];
    address_t addr_a{ array_a };
    address_t addr_b{ array_b };

    // ...
```


Addresses

- C++ has a solution to this:
  - A variable (local or parameter) is declared to be the "address of an int" or the "address of a double" by prefixing an asterisks to the identifier

```
int main() {
    int array_a[10];
    double array_b[10];

    int *addr_a{array_a};    // 'addr_a' is a variable
                             // that stores the address
                             // of an 'int'

    double *addr_b{array_b}; // 'addr_b' is a variable
                             // that stores the address
                             // of a 'double'
```


Address of variables

- Question: What about local variables or parameters?
  - They, too, must have addresses…how can we find those addresses?
  - The unary & operator returns the address of the operand

```
void f( int n );

void f( int n ) {
    double a;
    long b[10];

    std::cout << "The address of the parameter 'n' is      "
              << &n << std::endl;
    std::cout << "The address of the local variable 'a' is "
              << &a << std::endl;
    std::cout << "The address of the local array 'b' is "
              << b << std::endl;
}
```

```
The address of the parameter 'n' is      0x7fff725f073c
The address of the local variable 'a' is 0x7fff725f0798
The address of the local array 'b' is    0x7fff725f0740
```

Address of variables

- Similarly, we can assign these addresses to variables

```
void f( int n );

void f( int n ) {
    double a;
    long b[10];

    int    *p_n{ &n };
    double *p_a{ &a };
    long   *p_b{  b };

    std::cout << p_n << std::endl;
    std::cout << p_a << std::endl;
    std::cout << p_b << std::endl;
}
```

Output:

```
0x7fff725f073c
0x7fff725f0798
0x7fff725f0740
```


Pointers

- A variable that stores an address is referred to as a *pointer*
  - Suppose you see:

    `double *p_var{};`

    you may describe the variable `p_var` as either

    "a variable that stores the address of a `double`"

    "a pointer to a `double`"

- To be absolutely clear to the reader,

  all pointer identifiers will be prefixed with `'p_'`
  - This is a naming convention widely used in industry
  - We will use this naming convention in this course

Size of a pointer

- Question: You are compiling code for a processor and you ask yourself "How many bytes is an address on this processor?"
  - Solution:

    `std::cout << sizeof( int * ) << " bytes" << std::endl;`

- It doesn't matter which type you pick: `bool`, `int`, `double`, `short`
- On the computer *eceubuntu*, we get the output of 8 bytes or 64 bits
  - It is unlikely any of you will get anything else

    unless you have access to your parent's laptop or desktop

How do we access what is at that address?

- Suppose you have an address

  `int *p_datum{ &n };`
- Question: How do we access the integer stored at that address?
  - Solution: Prefix the identifier with an asterisk

```
int main() {
    int n{ 42 };
    int *p_datum{ &n };

    std::cout <<  p_datum << std::endl;
    std::cout << *p_datum << std::endl;
    *p_datum = 100;
    std::cout << *p_datum << " == " << n << std::endl;
    n = 99;
    std::cout << *p_datum << " == " << n << std::endl;
    return 0;
}
```

Output:

```
0x7fff725f073c
42
100 == 100
99 == 99
```

Why pointers?

- Why do we need to store addresses?
  - Arrays are fixed in their size:
    - What if you have to change an array?
  - Local variables and parameters are out of scope as soon as a function returns
    - What if we require memory that continues to exist outside the scope of a given function?

Making sense of C++ declarations

- Consider these declarations:

```
int n;
int array[10];
int *p_datum;
int gcd( int m, int n );
```

For `n`, to get an integer value, you must use `n`
For `array`, to get an integer value, you must use `array[ n ]`
For `p_datum`, to get an integer value, you must use `*p_datum`
For `gcd`, to get an integer value, you must call `gcd( n1, n2 )`

This was the original C design
Unfortunately, this doesn't work for `int &n;`

Capacity of an array?

- Up to now, we have indexed arrays with `int`:

```
char array[100];
for ( unsigned int k{0}; k < 100; ++k ) {
    array[k] = '\0';
}
```

- With a 64-bit computer, we could declare a much larger array:

```
char array[10000000000];
for ( unsigned int k{0}; k < 10000000000; ++k ) {
    array[k] = '\0';
}
```

  - Problem: the maximum `int` is $2^{32} - 1$ or approximately 4 billion

Capacity of an array?

- The type of an index into an array depends on the processor
  - For 8-bit processors, we could use `unsigned char`
  - For 32-bit processors, we could use `unsigned int`
  - For 64-bit processors, we should use `unsigned long`
- Fortunately, there is a universal solution:
  - The type `std::size_t` is always guaranteed to be an unsigned integer that can store the maximum index for a particular processor
- From now on,
  if the purpose of a local variable is to index into an array,
  it will be declared to be of type `std::size_t`

Summary

- Following this lesson, you now
  - Understand the concept of addresses and storing them
  - Know how to access the address of data in memory
  - Know how to access and manipulate data at a memory location
  - Understand the size of addresses on different computer architectures
  - You know about `std::size_t`

References

[1] https://en.wikipedia.org/wiki/Pointer_(computer_programming)

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.



Disclaimer