# Class 02

Memory, Control Structures

## Outline

- Variables & Memory
- Logical Statements & Control Structures
  - conditional blocks
  - while loops

## Variables

- To better understand variables, we need to dive a little deeper into the basic memory layout your computer uses.

- Data requires memory from the computer to store. The memory we are concerned about are RAM and cache memory.

- Whenever you create a variable or load data, it is stored in the computer's memory and assigned an address.

## Basic Memory Layout

| 0x014 | 0x015 | 0x016 | 0x017 | 0x018 | 0x019 | 0x01A | 0x01B | 0x01C | 0x01D |
|---|---|---|---|---|---|---|---|---|---|
| x = 'A' | y = 9.81 | | | | | | | | |

- Consider the following variables

```cpp
auto x = char{'A'};      // a char is 1 byte
auto y = double{9.81};   // a double is 8 bytes
```

## Basic Memory Layout

| 0x014 | 0x015 | 0x016 | 0x017 | 0x018 | 0x019 | 0x01A | 0x01B | 0x01C | 0x01D |
|---|---|---|---|---|---|---|---|---|---|
| x = 'A' | y = 9.81 | | | | | | | | |

- The green row represents memory addresses. Each byte in memory is given a unique hexadecimal address.
- The grey boxes are the actual memory blocks/data. Note that some data is larger than others, and so they take up more than one address!

## Basic Memory Layout

| 0x014 | 0x015 | 0x016 | 0x017 | 0x018 | 0x019 | 0x01A | 0x01B | 0x01C | 0x01D |
|---|---|---|---|---|---|---|---|---|---|
| x = 'A' | y = 9.81 | | | | | | | | z = true |

- When we add another variable, the following blocks are consumed.

```
auto x = char{'A'};      // a char is 1 byte
auto y = double{9.81};   // a double is 8 bytes
auto z = bool{true};     // a bool is 1 byte
```

## Variables & Memory

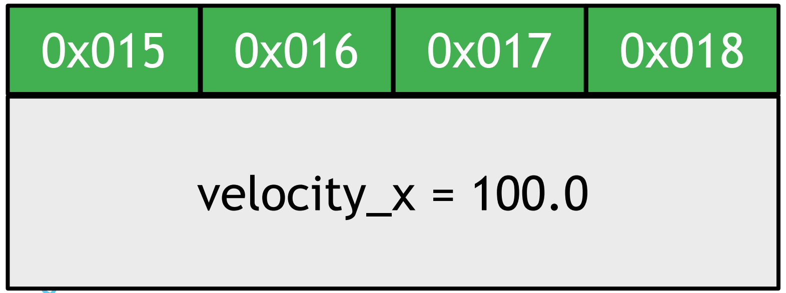
- Variables are generally small in memory.
  - An int is 4 bytes; our phones nowadays hold up to 32 gigabytes, that is 32 billion bytes!
- This does not mean that variables are free! There are costs outside of the raw memory they consume:
  - The CPU needs to find a place in memory to store the new variable
  - Then the memory needs to be reserved (allocated)
  - Then the memory needs to be cleaned up when it is no longer used
- It you are careless in managing your memory, you can create bottlenecks and micro-inefficiencies that can build up to something substantial over time!
  - 1 second may not sound like a lot of time, but compound that over 100,000 runs of your program, and you are now waiting an extra day for everything to complete!
- Memory layout strategies change from system to system.

# Overwriting Variables

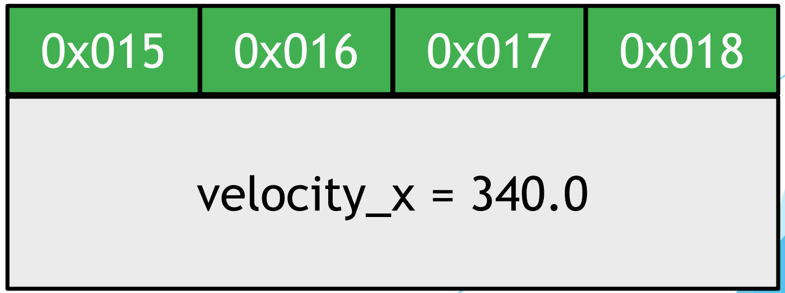
- Once we have a variable, we can change its value. In many cases it is because we no longer care about the old value. It is then more efficient to just reuse the variable rather than create a brand new one.

```
auto velocity_x = float{100.0};
if (boost)
{
    velocity_x = 343.0; // this is much better than creating a new
}                       // variable named boosted_velocity_x
```

Before



After



## Improper Use of Variables

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    cout << x << endl;
}
```

## Improper Use of Variables

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    cout << x << endl;
}
```

- We never **defined** x! What is it supposed to be? An int? A float? What is its value supposed to be?
- We cannot use variables that we have not defined!

## Improper Use of Variables

- What is wrong here? We even defined x this time!

```
#include <iostream>
using namespace std;
auto main() -> int
{
    cout << x << endl;
    auto x = int{9 + 18};
}
```

## Improper Use of Variables

- What is wrong here? We even defined x this time!

```
#include <iostream>
using namespace std;
auto main() -> int
{
    cout << x << endl;
    auto x = int{9 + 18};
}
```

- We tried using x **before it was defined**!
- C++ is like a strictly ordered TODO list. The computer interprets it top to bottom and when it is reading a line of code, it has no idea what is ahead.

## Improper Use of Variables

- What is wrong here? We defined x before using it!

```
#include <iostream>
using namespace std;
auto main() -> int
{
    int x;
    cout << x << endl;
}
```

## Improper Use of Variables

- What is wrong here? We defined x before using it!

```
#include <iostream>
using namespace std;
auto main() -> int
{
    int x;
    cout << x << endl;
}
```

- We never gave x a value!
- This is perfectly legal C++, *but it is bad C++*. This results in **undefined behavior, or UB.**
- This means that x can be any value (any value valid for an int).

## Improper Use of Variables

- What is wrong here? We defined x before using it!

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto x = int{};    // we can even let C++ deduce that this should be 0!
    cout << x << endl;
}
```

- We can ensure we never forget to initialize x by using our modern syntax!

## Logical Statements

- We have learned about the variable type **bool**, these are variables that logically are equivalent to **true** or **false**.
- To start we can think of a logical statement as being a comparison of one piece of data to another. We have the following C++ **operators** to make comparisons with.
  - == equality
  - != inequality
  - > greater than
  - < less than
  - >= greater than or equal to
  - <= less than or equal to

## Logical Statements

- What are the values of each of the following bool variables?

```
auto q = bool{7 < 10};
auto w = bool{0.1 == 0.2};
auto e = bool{1 > 0};
auto r = bool{100 != 1};
auto t = bool{'J' == 'j'};
auto y = bool{'J' != 'j'};
auto u = bool{-92 >= -94};
auto i = bool{3.1415 > 3.14};
auto o = bool{true == true};
auto p = bool{0};
auto a = bool{1};
```

## Logical Statements

- What are the values of each of the following bool variables?

```
auto q = bool{7 < 10};          // true
auto w = bool{0.1 == 0.2};      // false
auto e = bool{1 > 0};           // true
auto r = bool{100 != 1};        // true
auto t = bool{'J' == 'j'};      // false
auto y = bool{'J' != 'j'};      // true
auto u = bool{-92 >= -94};      // true
auto i = bool{3.1415 > 3.14}; // true
auto o = bool{true == true};  // true
auto p = bool{0};               // false
auto a = bool{1};               // true, for any non-zero number!
```

## Control Structures

- Every programming language provides a way to do different things under different conditions, or to assume a behavior while some condition holds true.
- These concepts are known as control structures. There are a number that exist, but for now we only care about the following two:
  - conditional-statement: perform some action if a given condition is true
  - while loop: perform some action *repeatedly* while some condition is true, until that condition is no longer true.
- Control structures are just constructions of logical statements and actions: *cause and effect*.

## Conditional Statements

- At their core, these are just *if x* … *then y* statements.
  - If the altitude is less or equal to 0, reverse the y-velocity
  - If the rate of growth exceeds 10 meters per second, apply a reduction algorithm
  - If the fuel tank has less than an ounce of gasoline, stop driving.

- C++ gives us a simple way to express a conditional statement:

```
auto power_level = double{9000.0001};
if (power_level > 9000.0)
{
    cout << "It is over 9000!" << endl;
}
```

## Conditional Statements

```
auto power_level = double{9000.0001};
if (power_level > 9000.0)
{
    cout << "It is over 9000!" << endl;
}
```

- Conditions must ultimately be a **bool** (true or false). What value does the condition take in this example?

- Does the action get executed? Why or why not?

# Conditional Statements

```
auto power_level = double{9000.0001};
if (power_level > 9000.0)
{
    cout << "It is over 9000!" << endl;
}
```

- Here the condition is:

```
power_level > 9000.0
```

- Here the action is:

```
cout << "It is over 9000!" << endl;
```

## Conditional Statements

- We will often need to support and check multiple conditions or provide alternatives when the initial conditional check is false.

- This is done using *else* and *else if* blocks. This would read as:

  "If X then do Y, else do Z"

  or

  "If X then do Y, else if A then do Z, else do B"

## Conditional Statements

```
auto distance = int{10};
if (distance < 5)
{
    cout << "too close!" << endl;
}
else if (distance < 20)
{
    cout << "just right!" << endl;
}
else
{
    cout << "too far!" << endl;
}
```

## Improper Use of Conditional Statements

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    if (false)
    {
        cout << "Hello World!" << endl;
    }
}
```

# Improper Use of Conditional Statements

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    if (false)
    {
        cout << "Hello World!" << endl;
    }
}
```

- Our conditional statement is *always false*, and so we will never execute the code inside of the block!

## Improper Use of Conditional Statements

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = int{0};
    cin >> a;
    if (a = 20)
    {
        cout << "Hello World!" << endl;
    }
}
```

## Improper Use of Conditional Statements

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = int{0};
    cin >> a;
    if (a = 20)
    {
        cout << "Hello World!" << endl;
    }
}
```

- Here we are not comparing a against 20! We need to check for equality, which is ==. Instead, we are assigning 20 to a, and then using a as the bool!

## Improper Use of Conditional Statements

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = int{10};
    else if (a > 5)
    {
        cout << "Hello World!" << endl;
    }
}
```

## Improper Use of Conditional Statements

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = int{10};
    else if (a > 5)
    {
        cout << "Hello World!" << endl;
    }
}
```

- We must start conditionals with an if!

## Improper Use of Conditional Statements

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = bool{true};
    if (a == true)
    {
        cout << "Hello World!" << endl;
    }
}
```

# Improper Use of Conditional Statements

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = bool{true};
    if (a)
    {
        cout << "Hello World!" << endl;
    }
}
```

- It is redundant to check the value of a bool in a conditional statement!
- If you have a bool, just use the bool directly, ***do not compare the value***.

## Compound Conditionals

- Lastly, we can join multiple conditionals together to form more complicated logical statements.

- && (*and)* can be used to check if two conditionals are true

- || (*or*) can be used to check if at least one conditional is true

```
if (a >= a && a <= 10) // checks if a is in the closed range [0, 10]
{

}
else if (a < 0 || a > 10) // check if a is not in the closed range [0, 10]
{

}
```

## While Loops

- Consider the following:

```
auto mass = double{100.0};    // kilograms
auto force = double{761.0};   // newtons
if (force / mass == 9.81)
{
    cout << "Object is in freefall" << endl;
}
```

- Our program will simply check if the force divided by the mass is equal to 9.81. If it is, this means the object is in freefall, and so we simply output that it is.

## While Loops

- In many contexts we need to repeatedly check if some condition is holding true.
  - For example, you may need to check to make sure some threshold is breached. So while the threshold has not been breached, continue reprocessing.
  - This can be seen in the algorithm for the producing the Mandelbröt Set. In that algorithm we are constantly computing a complex number and checking if its magnitude is less than 2.0.
- A while loop accomplishes this: while some condition is true, execute some action. When the condition is no longer true, proceed with the rest of the program.

## While Loops

- Example

```
auto n = int{0};
while (n < 10)
{
    cout << n << endl;
}
```

- What is this doing? What is the output?

## While Loops

- Example

```
auto n = int{0};
while (n < 10)
{
    cout << n << endl;
    n++;
}
```

- What is this doing? What is the output?

## While Loops

- Example

```
auto i = int{1};
auto n = int{10};
while (n > 1)
{
    i = n * i;
    n = n - 1;
}
cout << i << endl;
```

- What is this doing? What is the output?

## Improper Use of While Loops

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = int{10};
    while (a < 100)
    {
        cout << "too small!" << endl;
    }
}
```

## Improper Use of While Loops

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = int{10};
    while (a < 100)
    {
        cout << "too small!" << endl;
    }
}
```

- Our loop will never end! Our variable a is *always* less than 100, and so the loop will run indefinitely!

# Improper Use of While Loops

- This is better…

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = int{10};
    while (a < 100)
    {
        cout << "too small!" << endl;
        a = a + 25;
    }
}
```

- Now a grows by 25 until it is no longer less than 100! How many times does the cout statement get executed? What is a's final value?