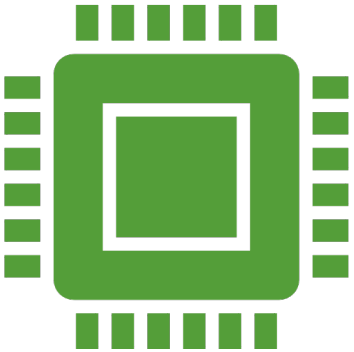


ADVANCED GAMEPLAY PROGRAMMING

WEEK 4

ADVANCED C#

DATA

- What makes up a variable?
  - Access modifier (can be implied)
  - Type
  - Name
  - Value

TYPES IN C#

- Everything has a type in C#
- Type describes the form the data inside the variable takes
- There are two kinds of types:
  - Reference types
  - Value Types

VALUE TYPES

- Simple Types
  - integral
  - floating point
  - char
  - boolean
- Enumerated types (enum)
- Struct (Composite Type)

INTRO TO BINARY

BUILT IN INTEGRAL TYPES

C# type	Values	Size	Signed
sbyte	-128 to 127	1 byte	yes
short	-32,768 to 32,767	2 bytes	yes
int	-2,147,483,648 to 2,147,483,647	4 bytes	yes
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8 bytes	yes

short a = 12

00000000 00001100

BUILT IN INTEGRAL TYPES

C# type	Values	Size	Signed
byte	0 to 255	1 byte	no
ushort	0 to 65,535	2 bytes	no
uint	0 to 4,294,967,295	4 bytes	no
ulong	0 to 18,446,744,073,709,551,616	8 bytes	no

byte a = 140

10001100

FLOATING POINT TYPES

C# type	Size	Precision	Range
float	4 bytes	7 digits	$+-1.5 \times 10^{-45}$ to $+-3.4 \times 10^{38}$
double	8 bytes	15-16 digits	$+-5.0 \times 10^{-324}$ to $+-1.7 \times 10^{308}$
decimal	16 bytes	28-29 decimal places	$+-1.0 \times 10^{-28}$ to $+-7.9 \times 10^{28}$

OTHER VALUE TYPES

- Char is 2 bytes (65,536 possible letters)
- A boolean is a byte (256 possible values?)
- Enumerated types (enum)
  - ints by default
  - Can inherit from any integral type:

```
enum Player : byte {
    Undefined = 0,
    First,
    Second,
    Third,
    Fourth
};
```

- Struct (Composite Type)
  - Size of things they're composed from

VALUE VS. REFERENCE

- Usually using the word “new” is an indicator
  - Creates value for the object, and assigns the variable with a *reference* to that value
- Passing by Value
  - The system creates a separate copy of a variable in another method, so that if value got changed in the one method won't affect on the variable in another method.
- Passing by Reference
  - The system passes a reference to variable in another method, so that if value got changed in the one method it’s changed in the other method

Stack

This is where the program code is, as well as the currently running function.

- Organized
- Fixed size – can run out of space
- Data guaranteed to be there.
- Data only exists until function ends
- Very cheap to compute with

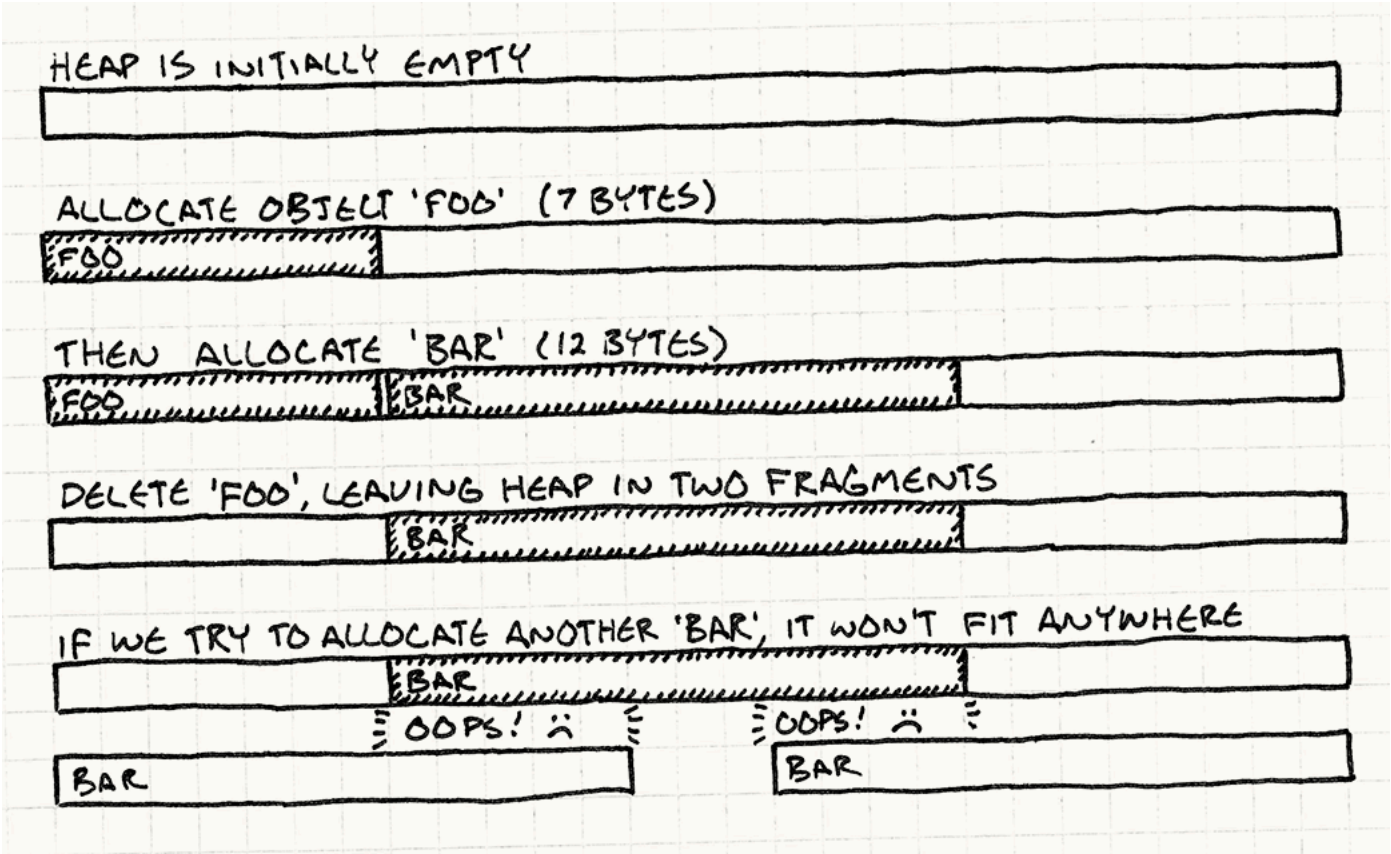
```
var x = 10;
```

Heap

This is where the data that is referenced is.

- Disorganized
- No fixed size
- No guarantee the data is there.
- Data persists after function ends
- Expensive to compute with

```
var c = new List<int>();
```



https://gameprogrammingpatterns.com/object-pool.html

REFERENCE TYPES

- Instead of **data**, a reference type holds a **pointer** to where the data is stored.

- Arrays (Composite Type) (These are reference types *even if the array is of value types*.)
- Interface Types
- Delegate Types
- Classes (Composite Type)

BASICS OF BINARY

- Value
- Amount of information
- Bitwise Operators
  - &
  - |
  - << , >>

SO WHAT IS A DELEGATE?

- A delegate is a reference type.
- The value it refers to is a method.
- Usually, when you call a function, you call it directly.
- Sometimes, you want something else to call the function, or call it later

BUILT IN DELEGATE TYPES

- Actions

  Action<somenumber, ofTypes> identifier
- Funcs

  Func<someNumber, ofTypes, typeReturned> identifier
- Predicates

  Predicate<Input> identifier

```
public abstract class NPC {
    public Action movement;
    public Action<string> speak;
    public Action<Player, int> dealDamage;

    public Func<Item> receiveItem;
    public Func<Vector3, bool> attemptTeleport;

    public Predicate<Item> hasItemOfType;
}
```

```
public void MoveWithoutGravity() {

}

public void UnderwaterMovement() {

}

public void Walking() {

}

private void LoadSpaceLevel() {
    foreach (var npc in allNPCs) {
        npc.movement = MoveWithoutGravity;
    }
}
```

CREATING YOUR OWN DELEGATE TYPES

- You can specify your own w/ the delegate keyword:

  delegate result-type identifier ([parameters]);

```
delegate void Handler(AGP_Event e);

public void OnGoalScored(AGP_Event e) {

}
```

EXAMPLE CODE

```
delegate float DoMath(float a, float b);

public float Sum(float a, float b) {
    return a + b;
}

public float Divide(float a, float b) {
    return a / b;
}

public void Start() {
    DoMath mathOperation = new DoMath(Sum);

    Debug.Log(mathOperation(3, 2)) // prints 6

    mathOperation = Divide;

    Debug.Log(mathOperation(3, 2)) // prints 1.5
}
```

HOW DO YOU USE DELEGATES?

- Allow you to *make decisions at runtime*

- Picking one of multiple functions
- Giving a different class a function to run
- When you want to use multicast
- When you want to encapsulate a function to run in the future, or run multiple times in the future.
- When you want to encapsulate a function to undo it.

EXAMPLE CODE:

```
public bool saveToFile = false;

delegate void DisplayInformation (string toDisplay);

public void ConsoleLogInfo(string toDisplay) {
    Debug.Log(toDisplay);
}

public void WriteOutInformation(string toDisplay) {
    var fileName = GetFileName();
    var path = Application.dataPath + fileName;

    using (var fs = new FileStream(path, FileMode.Create)) {
        using (var writer = new StreamWriter(fs)){
            writer.Write(currentPuzzle);
        }
    }
}
```

EXAMPLE CODE:

```
DisplayInformation currentWayToDebug;

public void Start() {
    if (saveToFile) {
        currentWayToDebug = WriteOutInformation;
    }
    else {
        currentWayToDebug = ConsoleLogInfo;
    }
}

public void DebugInfo(string toDebug)
{
    currentWayToDebug(toDebug);
}
```

STRATEGY PATTERN

```
delegate void Attack(Enemy e);
class Player
{
    private Attack _attack;

    public Player(Attack attack)
    {
        _attack = attack;
    }

    public void Update()
    {
        var e = GetNearestEnemy();
        if (e !== null) _attack(e);
    }
}

// Somewhere else...
Attack attack = ChooseCustomAttackBasedOnRuntimeFunction();
Player player = new Player(attack);
```

WHAT ARE CALLBACKS?

- Callbacks are passing delegates to another system to be run based on future state

```
public IEnumerator GetPlayerDetailsFromServer(Action onSuccess, Action onFailure) {
    yield return new MakeServerRequest();

    if (playerDetails == null)
        onFailure();
    else
        onSuccess();
}

// At start of game:
GetPlayerDetailsFromServer(StartMatch, ShowCouldntConnect);

// In game:
GetPlayerDetailsFromServer(UpdateEnemyPosition, DisconnectFromServer);
```

WHAT ARE LAMBDAS?

- A lambda is the *data* of a function.
- Like any data type, you can use the data with out giving it a name.

```
// At start of game:
GetPlayerDetailsFromServer(StartMatch, () => Debug.Log("Couldn't connect."));
```

- They can have parameters:

```
var x = (int someNumber, float ofParameters) => {
    // What happens in the lambda expression
};
```

WHAT ARE TYPE GENERICS?

- A "generic" is a way of defining **methods and classes** with a placeholder parameter or return **type**

```
public void SpawnEnemy<T>(Vector3 location) {
    // create an enemy of Type T
}

SpawnEnemy<FireImp>(portal);
SpawnEnemy<Toucan>(tree);
```

```
public T MakeEnemy<T>(string EnemyName) where T : Enemy {
```

```
public class Portal<T> where T : ShipType {

}
```

WHY USE TYPE GENERICS?

- Reusability
- Type Safety
- Performance

COMPLEXITY

COMPLEXITY

- Amount of resources required to solve a problem
  - Can be time (time complexity)
  - Can be amount of space (space complexity)
  - Because computers are different - time is generally expressed as the number elementary operations
- Average complexity vs. Worst-case complexity
- "Big O notation"
- As the amount of needed resources varies with the input, a problem's complexity usually includes the "n" difficulty of solving a problem
- Evaluating the complexity of an algorithm is an important part of algorithm design, as this gives useful information on the performance that may be expected.

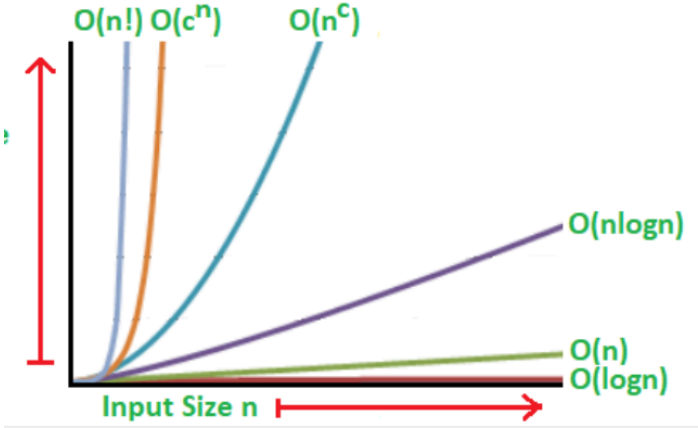
COMPLEXITY EXAMPLES

Here's some examples of problems that (naively) are certain difficult.

- Constant Time - O(1) – takes one step to solve every time.
  - Is the player swimming?
  - Does the ball exist?
  - What's the current score?
- Linear Time - O(N) – takes at most N steps to solve for N inputs
  - Which of these enemies is closest to the player?
  - Which of these players has the ball?
  - Which of these grid-spaces are 1 away from this tile?
  - How many enemies are in critical health?
- Quadratic Time $O(N^2)$ – takes at most N * N steps to solve for N inputs
  - All tiles groups of tiles that are adjacent to each other.
  - Does this list contain duplicates?

COMPLEXITY EXAMPLES (CONT.)

- Logarithmic Time O(log N) – takes at most $\log_2 N$ steps to solve for N inputs
  - Searching through a sorted list
  - Things you can divide and conquer
  - Using trees to solve certain kinds of problems
- Superlinear Time – O(NlogN)
  - Usually, the difficulty of sorting something
- Exponential Time – $O(2^N)$ and Factorial Time - O(N!)
  - Too slow to be useful.



COMPLEXITY FOR GAMES

- Making a game is like making 100s of algorithms from scratch
- Identifying inefficient use of resources is an important part of optimizing your games
- Identifying which elements of your game cannot be solved perfectly quickly is important for knowing how to design AI

ASYMPTOTIC ANALYSIS

- Analyzing how optimal a solution to a problem is can be difficult / time consuming / impossible
- You can't always just "run the game and see which is faster"

CONSTANT TIME

- Assigning a value to a variable
- Looking up the value of a particular element in an array
- Comparing two values
- Incrementing a value
- Basic arithmetic operations such as addition and multiplication

- Assume branching (the choice between if and else parts of code after the if condition has been evaluated) occurs instantly and don't count those instructions

LOOPS

- Loops that take constant time irrespective of input are O(1)
- Loops that happen based on input length are O(N)
- Nested loops based on input are $O(N^C)$, where c is the number of nested loops
- If the loop variable is divided by a constant amount (you disregard multiple inputs each loop), it's O(logN)

- Simple programs can be analyzed by counting the nested loops of the program

```
public bool ContainsDuplicates() {
    foreach (var item in inventory) {
        foreach (var item in inventory) {

        }
    }
}
```

ALSO…

- Instantiating anything on the heap is slow (anything w/the ‘new’ keyword)
- Searching is slow
- Sorting is slow
- Only pay attention to the slowest part of an algorithm – if most of the algorithm is constant time, but there’s one triple-nested loop, the problem is the triple-nested loop.