WEEK 2

Foundational Ideas &

Dynamic Memory

AGENDA

- Week 2-1
  - Types,
  - Declarations and Definitions
  - Header File Includes Ordering
  - Scope and Function Overloading
  - References and Array of Pointers
- Week 2-2
  - Dynamic Memory

WEEK 2-1

Types, Declarations, Scope, Overloading, References, Array of Pointers

| Reviewing old and new ideas

TYPES

- As C++ is and extension of the C programming language, the types that can be used in C++ are inherited from C with some additions
- These fundamental types include:
  - char
  - int (short, long, long long)
  - float (double, long double)
  - bool – Note bool is not available in C

TYPES - BOOL

- The bool type is used to store a logical true/false value. Eg bool x = true;
- The value of a bool type can be inverted with the use of the ! negation operator
  - !x – If x is true then it is now negated to be false.

- Conversions from bool to int and vice versa generally produce values of 0 and 1. Wherein false is equated to 0 and true is relegated to 1 (however any non-zero number is considered to be true)

TYPES

- Compound types are types that are composed of other (fundamental) types
- You may already know these from C as structs
- In C++ there are also classes which we’ll delve into in the following weeks

TYPES - AUTO

- A keyword that was introduced in C++11, the 2011 standard of the language is the auto
- auto is used in place of a type when initializing a variable
  - int x = 3; vs auto x = 3;
- The auto keyword is able to determine the type of the variable x through the value given from the right
  - auto x = 3; // 3 is an integer value so the type of x is also an int
- auto is useful in that we don’t need explicitly specify the type of things if it can be determined by the compiler

DECLARATIONS

- Declaration is the process of assigning a type to an entity (or a name). The entity can be a variable, an object, a function…
- The type we assign to it allow for the C++ compiler to understand what sort of meaning lies with this entity and how it should be interpreted
- An example of a declaration could be this function:
  - double minus(double a, double b);
  - This declares a function that perhaps performs subtraction between two double numbers and returns the result
  - Note that it doesn't have a definition yet.

DECLARATIONS

- There is a certain type of declaration called the forward declaration
- This sort of declaration is statement to indicate to the C++ compiler of an entity that will be used but is perhaps not yet defined in full.
- The forward declaration is like a prototype in the header file

DEFINITIONS

- Definitions are the meanings or implementation of a declared entity such as the contents of a function.
- From our previous minus function its definition could be as follows:

```
double minus(double a, double b) {
                    return a – b;
}
```

- As perhaps alluded to previously in our stint with namespaces there is the idea of a one definition rule where a definition may only occur once with in a particular scope

DECLARATIONS AND DEFINITIONS

- Declarations aren’t necessarily definitions as the former doesn’t associate meaning to an entity while the latter does.
- When considering the declaration and definitions of entities in our programs we mainly want to structure them in a way that avoids breaking the one definition rule
- This idea will lead to thinking about how we include header files and perhaps where we should be placing our definitions

DECLARATIONS – DEFAULT PARAMETER VALUES

- If desired, function parameters can have default values to them if they aren't supplied
- The syntax for it looks like the following:

```
double minus(double a, double b = 5.0);
```

- The parameters that have default values have to be placed on the most right side of the parameter list.
- The function can now then be called without specifying anything for the b variable:
  - `double x = minus(4.0) // One parameter; a = 4.0, b = 5.0`
  - `double x = minus(4.0, 1.0) // Two parameters; a = 4.0, b = 1.0`

HEADER FILE INCLUDE ORDERING

- When including header files in any portion of your code there is a standard ordering that should be adhered to:
  1. Include system libraries / headers first – eg #include <iostream>
     - System libraries are typically enclosed by the <> symbols
  2. Include other libraries / headers second – eg #include "libx"
  3. Include your own libraries / headers last – eg #include "mylib"
  4. After all these includes is when you can specify namespaces and other directives – eg using namespace …

SCOPE

- The idea of scope equates to the portion of code where a declaration is visible
    - Eg. given a variable of x from what portions of code can x be seen/used?

- There are a few different levels of scoping:
    - Global scope – visible to the whole program
    - File scope – visible to just the code within a file
    - Function scope – visible to the length of a function
    - Class scope – visible to the functions of the class
    - Block scope – visible to the code block
- The scope of a declaration can be said to be local to one of the above levels

SCOPE EX 1

```
#include <iostream>
using namespace std;

int main(){

int x = 3;
  cout << "What is x1: " << x << endl;

{ // This starts a ‘block’
int x = 5;
cout << "What is x2: " << x << endl;
} // This ends a ‘block’

cout << "What is x3: " << x << endl;
}
```

What is x during these print outs?

SCOPE EX 2

```
#include <iostream>
using namespace std;

int main(){

for (int x = 0; x < 10; x++){
  cout << x;
} // x goes out of scope here at the end of the block

cout << x << endl; // x can no longer be accessed here
return 0;
}
```

SCOPE EX 3 - SHADOWING

```
#include <iostream>
using namespace std;

int main(){

int x = 22;
for (int x = 0; x < 10; x++){
  cout << x << endl; // this x local to the scope of the loop ‘shadows’ the other x
} // This inner x goes out of scope here at the end of the block

cout << x << endl; // prints 22 here at the end
return 0;
}
```

FUNCTION OVERLOADING

- Having already encountered the issue of conflicting identifiers and the one definition rule, we would like to avoid it as much as possible
- However what if we wanted to reuse the same function names but have slightly different definitions for them
- C++ offers the feature of overloading these functions, functions with multiple definitions/meanings are thus called overloaded functions.

FUNCTION OVERLOADING

- A function declaration is comprised of a few different parts and the formation of these parts can be called the function's signature.

- The function’s signature is made of the following:
  - The function’s identifier or name
  - The parameter types
  - The order of the parameters

- Functions with the same identifier or name but a different signature will identify them as overloaded functions.
- A function’s return type and parameter names are not considered part of its signature

FUNCTION OVERLOADING

Function identifier

Parameter types

Order of parameters

```
int add(int a, int b);

int add(int a, int b, int c);
```

These two are overloaded functions.

REFERENCES

- In C++ a reference is an alias to the variable or object.
- It's similar to a pointer in that it refers to something but it has a few big differences:
  - Pointers can be made to point to something, nothing and can switch what it refers to, references will only refer to the original entity and can not be changed after initialization
  - References can't be null
  - References must be initialized when created.
  - References are declared using the & symbol eg. int & x = y // x is a reference to y

- We'll be mainly looking at references in a context of passing a variable into a function call
- Passing variables by reference serves as an alternative to passing them by address/pointers.

```cpp
// Swapping values by address
// swap1.cpp
#include <iostream>
using namespace std;
void swap ( int *a, int *b ); // Diff declar

int main ( ) {
int left;
int right;

cout << "left is " << left << endl;
cout << "right is " << right << endl;

swap(left, right);

cout << "After swap, left is " << left << "\n"
<< "right is " << right << endl;
}

void swap ( int *a, int *b ) {
int c;
c = *a; // Note the differences here
*a = *b;
*b = c;
}
```

```cpp
// Swapping values by reference
// swap2.cpp
#include <iostream>
using namespace std;
void swap ( int &a, int &b ); // Diff declar

int main ( ) {
int left;
int right;

cout << "left is " << left << endl;
cout << "right is " << right << endl;

swap(left, right);

cout << "After swap, left is " << left << "\n"
<< "right is " << right << endl;
}

void swap ( int &a, int &b ) {
int c;
c = a; // Note the differences here
a = b;
b = c;
}
```

WEEK 2-2

Dynamic Memory

| Getting more resources as needed

MEMORY

STATIC MEMORY

- Static memory refers to the amount of memory allocated for a program at load time.
- This amount of memory is determined at compile time
    - In other words of the memory given to a program is already set by the time it's run
- Memory that's used by a program needs to be returned to the system when that program has finished its execution. Memory that's statically allocated is done so automatically when the program exits / its code paths go out of scope.
- In our example of playdoh, perhaps we are given one jar of it every play time and that same jar is collected at the end of play time.

STATIC MEMORY

Playdoh Supply and Children		Playdoh On Loan
Playdoh Jar 1	Child 1	Child 1 – Has Jar 1
Playdoh Jar 2	Child 2	Child 2 - Has Jar 2
Playdoh Jar 3	Child 3	Child 3 - Has Jar 3

Static memory has the notion that what you defined and allocated at compile time is all the memory you have access to. In this example we have one child attached to on playdoh jar. This is fine if you always know how much you need ahead of time.

STATIC MEMORY

Playdoh On Loan	Playdoh Returned after Play time	
Child 1 – Has Jar 1	Playdoh Jar 1	Child 1
Child 2 - Has Jar 2	Playdoh Jar 2	Child 2
Child 3 - Has Jar 3	Playdoh Jar 3	Child 3

After play time is concluded every child returns their playdoh. So during the next play time they can be used again.

DYNAMIC MEMORY

- In the case where we perhaps don't know how much memory we need for our program's needs until runtime (if for example we require a user's input to decide), how are we able to request more resources at that time?
- This is where the notion of dynamically allocated memory comes into play. The ability to on demand request more memory from the system during run time.

DYNAMIC MEMORY

Playdoh Supply and Children

Playdoh Jar 1	Child 1
Playdoh Jar 2	Child 2
Playdoh Jar 3	Child 3

Playdoh On Loan

Child 1 – Has Jar 1

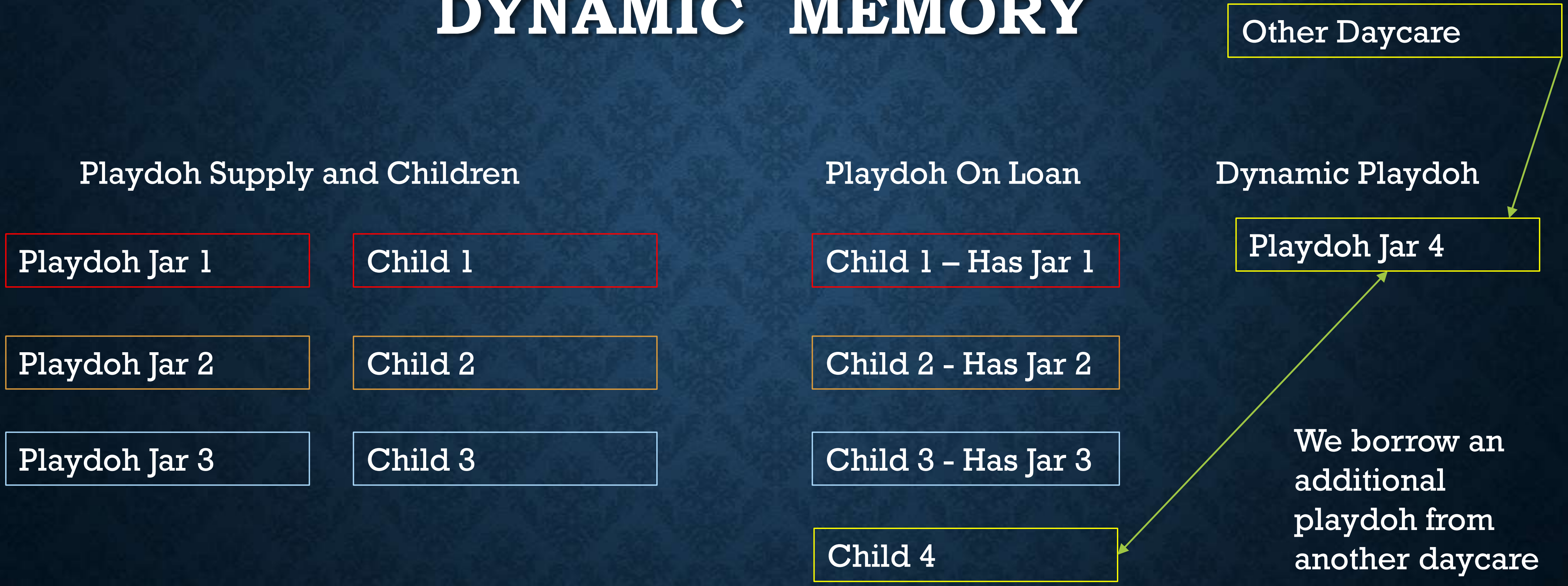
Child 2 - Has Jar 2

Child 3 - Has Jar 3

Child 4

A temporary child enters our daycare just for today after all the playdoh has been assigned. Will they not have any to play with?

DYNAMIC MEMORY



We borrow an additional playdoh from another daycare

DYNAMIC MEMORY

- The main key word in C++ that allows for dynamic memory allocation is new
- The typical syntax for this process makes use of pointers:
  - *pointer = new Type [size]*

- Eg. myStruct* s = new myStruct [5]; // allocating an array of myStructs
- Eg. myStruct* myint = new myStruct; // allocating a single instance

DYNAMIC MEMORY

- Going back to the playdoh example. We've borrowed/allocated one additional dynamic playdoh for use just for today. What happens to it after the day is done? Do we keep it with us?
- If we keep it with ourselves then the place, we borrowed it from will be one short. We only needed it for this one instance so we should return it back.
  - It is our responsivity to return this ourselves. We borrowed it after all.
  - If we don't no one else will be able to access that jar and we reduce the amount of playdoh everyone can use/share with one another

DYNAMIC MEMORY

Playdoh On Loan

Child 1 – Has Jar 1

Child 2 - Has Jar 2

Child 3 - Has Jar 3

Child 4

Dynamic Playdoh

$$\begin{tikzcd}\boxed{\text{Playdoh Jar 4}} \arrow[r] & \boxed{\text{Other Daycare}}\end{tikzcd}$$

We return Jar 4 back to the other daycare

DYNAMIC MEMORY

Playdoh On Loan

Child 1 – Has Jar 1

Child 2 - Has Jar 2

Child 3 - Has Jar 3

Child 4

Playdoh Supply and Children

Playdoh Jar 1	Child 1
Playdoh Jar 2	Child 2
Playdoh Jar 3	Child 3

Returns back to their own daycare after today

DYNAMIC MEMORY

- Compared to the new keyword when dealing with allocating dynamic memory there is also the delete keyword for deallocating memory when we are done with it.
- The issue of not returning memory to the system is that we reduce the amount of memory that’s available to the system overall
- The act of never returning this memory is considered a memory leak
- Similar to the new keyword the syntax for deallocating is:
  - *delete [] pointer;*

- Eg. delete [] myint; // the [] specifies we are deallocating an array
- Eg. delete myint; // without the [], we are deallocating a single instance