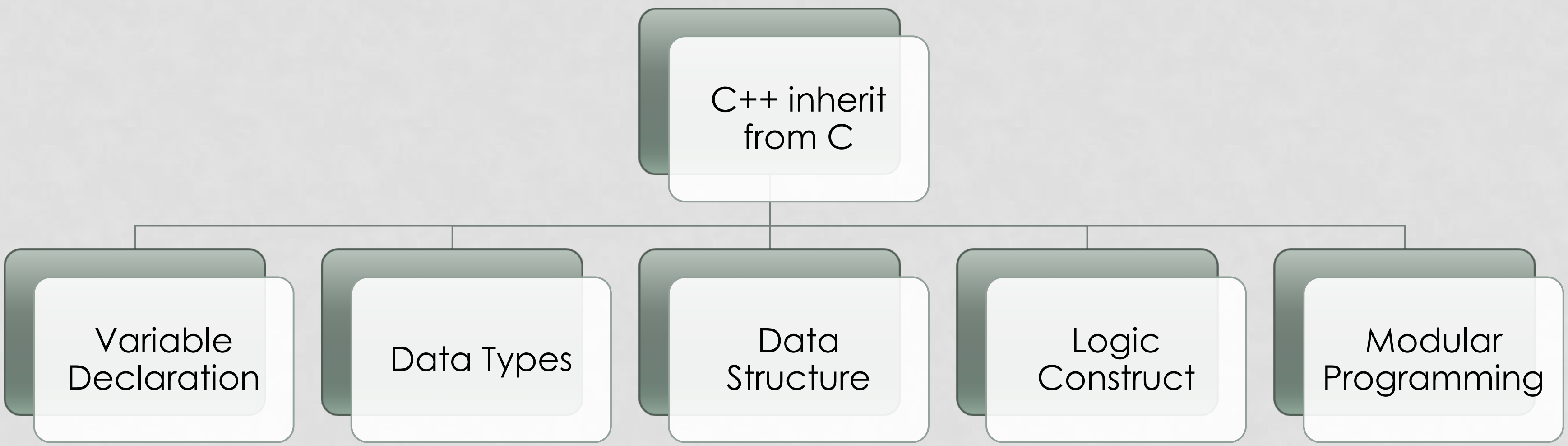
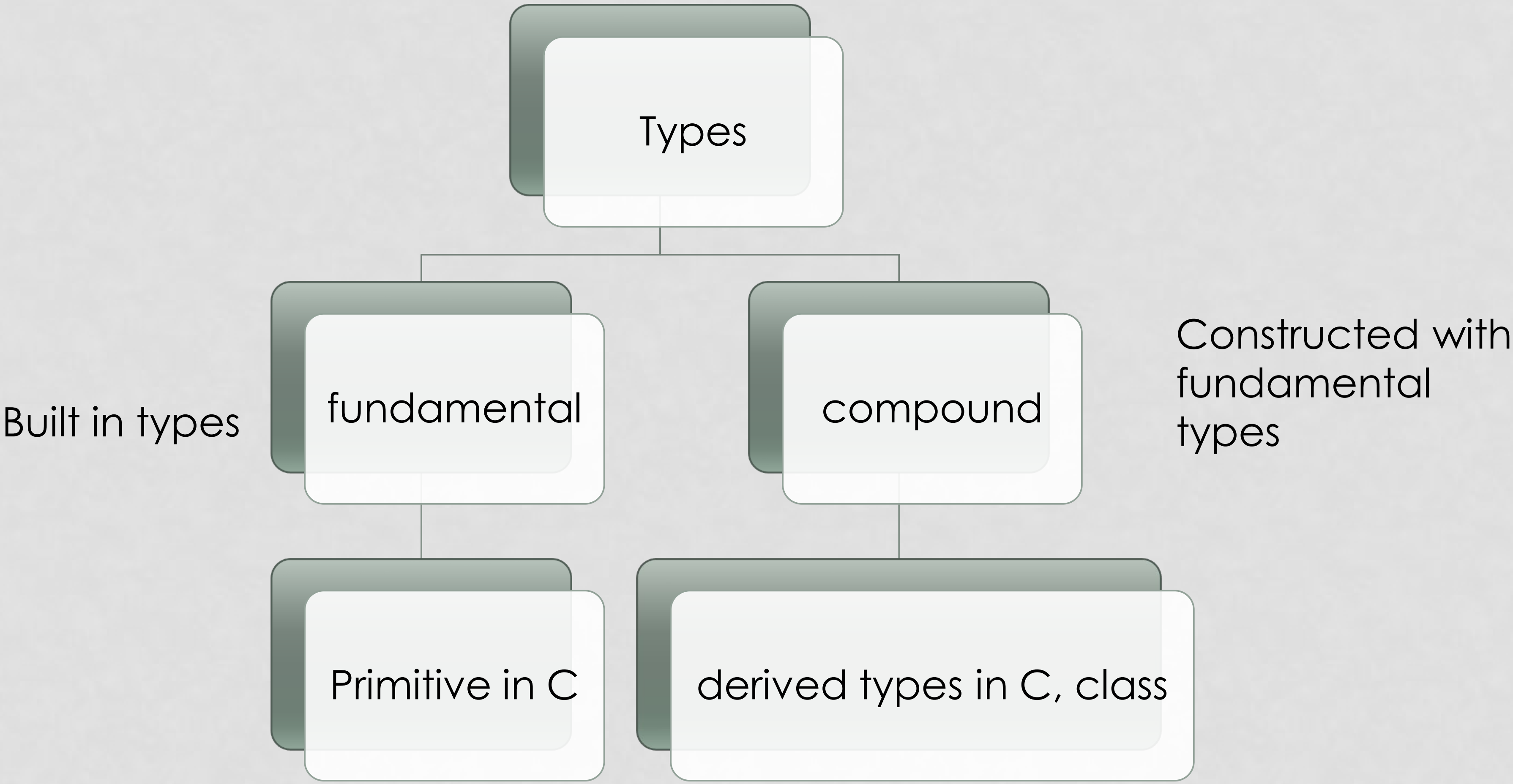
TYPES, OVERLOADING, REFERENCES, DYNAMIC MEMORY

FOUNDATION

TYPES


C++ inherit from C
Variable Declaration
Data Types
Data Structure
Logic Construct
Modular Programming

C++ TYPES


Types
Built in types
fundamental
compound
Constructed with fundamental types
Primitive in C
derived types in C, class

FUNDAMENTAL

- bool – not available in C
- char
- Int- short, long, long long
- float
- double – long double

bool to int

- The bool type stores a logical value : true or false
- The **!** operator reverses value:
  - **!true** is **false** and **!false** is **true**.
- The ! operator is self-inverting on bool, but not self-inverting on other type
- ! 0 = 1
- ! n = 0 , here n is any integer, except zero (0)
- // produces a value of 0

```
int x = 4;
cout << !x; // what is the output? 4 or 0 or 1
cout << !!x;    // what is the output? 4 or 0 or 1
```

COMPOUND TYPES

- A compound type is a type composed of other types
- Example

  struct, class

```
• // Modular Example
• // Transaction.h
struct Transaction {
        int acct; // account number
        char type; // credit 'c' debit 'd'
        double amount; // transaction amount
};
```

STRUCT IN "C" VS "C++"

```
// Modular Example - C++
// Transaction.h

struct Transaction {
    int acct;
    char type;
    double amount;
};
void enter(Transaction*);
void display(const Transaction*);
// ...

int main() {
    Transaction tr;
    // ...
}
```

```
// Modular Example - C
// Transaction.h

struct Transaction {
    int acct;
    char type;
    double amount;
};
void enter(struct Transaction*);
void display(const struct Transaction*);
// ...

int main() {
    struct Transaction tr;
    // ...
}
```

STRUCT IN C VS C++

- In C++ struct behaves like a class with some exceptions (default access modifier is public)
- C has no class feature, so, do the struct.
- In C there is no function inside the structure while in C++ we can define function
- We can have both pointers and references to struct in C++, but only pointers to structs of C are allowed

AUTO KEYWORD

- This keyword deduces the object's type directly from its initializer's type.

- We must provide the initializer in any **auto** declaration.

For example,

```
auto x = 4; // x is an int that is initialized to 4
auto y = 3.5; // y is a double that is initialized to 3.5
```

auto is quite useful: it simplifies our coding by using information that the compiler already has.

DECLARATIONS -DEFINITIONS

- The C++ language distinguishes between declarations and definitions and stipulates the one-definition rule.
- To avoid conflicts or duplication, we need to design our header and implementation files accordingly.
  - Modular programming can result in multiple definitions

DECLARATION

- Associating an entity (variable, object, function) with a type

```
int add(int , int )
```

- We have declared the add() function but did not specify any meaning to it.

```
struct Transaction;
```

- Forward declaration, something like function prototype (we have declared it but not defined yet)

DEFINITION

- A definition is a declaration that associates a meaning

```
// definition of display

int add(int a, int b) {
        int sum = 0;
        sum = a + b ;
        return sum;
}
```

- In the C++ language, a definition may only appear once within its scope
  - One definition rule

COMPARISON

- Forward declarations and function prototypes are declarations that are not definitions
- Associate an identifier with a type, but do not attach any meaning to that identifier.
- We may repeat such declarations several times within the same code block or translation unit.

PROPER HEADER FILE INCLUSION

- #include < ... > - system header files
- #include " ... " - other system header files
- #include " ... " - your own header files

  - We insert namespace declarations and directives after all header file inclusions.

SCOPE

- The *scope* of a declaration is the portion of a program over which that declaration is visible

  - global scope - visible to the entire program
  - file scope - visible to the source code within the file
  - function scope - visible to the source code within the function
  - class scope - visible to the member functions of the class
  - block scope - visible to the code block

GOING OUT OF SCOPE

```
for (int i = 0; i < 3; i++) {
    int j = 2 * i;
    cout << "The value of j is " << j << endl;
}
```

// j goes out scope just before the end of current iteration .j goes out of scope with each iteration

// i out of scope after completion of the iteration

SHADOWING

- **Variable shadowing** occurs when a variable declared within a certain scope (decision block, method, or inner class) has the same name as a variable declared in an outer scope

SHADOWING

```
// scope.cpp

#include <iostream>
using namespace std;

int main() {
    int i = 6;
    cout << i << endl;
    for (int j = 0; j < 3; j++) {
        int i = j * j;
        cout << i << endl;
        }
    cout << i << endl;
}
```

FUNCTION OVERLOADING

- You can have multiple definitions for the same function name in the same scope.

- The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. Function Signature

- You cannot overload function declarations that differ only by return type.

OVERLOADING-EXAMPLE

```
#include <iostream>
using namespace std;
class printData {
   public:
    void print(int i) {
    cout << "Printing int: " << i << endl;  }
    void print(double  f) {
   cout << "Printing float: " << f << endl;
}
   void print(char* c) {
  cout << "Printing character: " << c <<
endl;
      }
};
```

```
#include <iostream>
  int main(void) {
  printData pd;

  // Call print to print integer
  pd.print(5);
  // Call print to print float
  pd.print(500.263);

  // Call print to print character
  pd.print("Hello C++");

  return 0; }
```

When the above code is compiled

Printing int: 5

Printing float: 500.263

Printing character: Hello C++

FUNCTION OVERLOADING

- Same function name (identifier) multiple meaning
- Different parameters and their orders

```
 // Overloaded Functions
// overload.cpp
#include <iostream>
using namespace std;

// prototypes
void display(int x);
void display(const int* x, int n);

int main() {
    auto x = 20;
    int a[] = {10, 20, 30, 40};
    display(x);
    display(a, 4);
```

```
}
// function definitions
 //
void display(int x) {
    cout << x << endl;
 }

void display(const int* x, int n) {
    for (int i = 0; i < n; i++)
        cout << x[i] << ' ';
    cout << endl;
 }
```

DEFAULT PARAMETER VALUES

if the function logic is identical in every respect except for the values received by the parameters

```
// Default Parameter Values
// default.cpp

#include <iostream>
using namespace std;

void display(int, int = 5, int = 0);

int main() {

    display(6, 7, 8);
    display(6);
    display(3, 4);
}
```

```
void display(int a, int b, int c) {
    cout << a << ", " << b << ", " << c
<< endl;
}
```

PROTOTYPES

- A programming language may require a function declaration before any function call for type safety
- The declaration may be either a prototype or the function definition itself

```
// Compiler Error
int main() {
printf("Hello C++\n");
}
//with prototypes
#include <cstdio> // the prototype is in this header file
using namespace std;
int main() {
printf("Hello C++\n");
}
```

REFERENCE

- Reference is an alias or an alternate name of an existing variable.
- Contains same address and values
- & has different meaning in declaration (reference variable value) and expression (address-of variable)
- It is mainly used to support pass-by-reference in function call

REFERENCE

- type& newName = existingName;
- For example,

suppose we have the following example –

```
int i = 17;
```

We can declare reference variables for i as follows.

```
int&  r = i;
```

REFERENCE-EXAMPLE

```
include <iostream>
 using namespace std;

 int main () {
 // declare simple variables
         int i;
         double d;
// declare reference variables

       int& r = i;
       double& s = d;

   i = 5;
cout << "Value of i : " << i << endl;
cout << "Value of i reference : " << r << endl;

   d = 11.7;

cout << "Value of d : " << d << endl;
cout << "Value of d reference : " << s << endl;
return 0;  }
```

When the above code is compiled it produces the following result −

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7
```

HOW REFERENCING WORKS?

- Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.
- It is like pointer except
  - It is a name constant of an address
    - (once a reference is established you can not reference it to any other memory)
- Reference variables
  - Can not be null
  - Can not be changed once initialized to one object to another
  - Must be initialized
- Example: Note

ARRAY OF POINTERS

- Arrays of pointers are data structures like arrays of values.
- Arrays of pointers contain addresses rather than values.
- We refer to the object stored at a particular address by dereferencing that address
- An array of pointers provides an efficient mechanism

  for processing the set

ARRAY OF POINTERS

- Example, which makes use of an array of 3 integers −

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};

   for (int i = 0; i < MAX; i++) {

      cout << "Value of var[" << i << "] = ";
      cout << var[i] << endl;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

ARRAY OF POINTERS

- There may be a situation, when we want to maintain an array, which can store pointers to an int or char or any other data type available
- array of pointers to an integer −

```
int *ptr[MAX];
```

- This declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value.

ARRAY OF POINTERS

```
//Array of pointers example
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};
   int *ptr[MAX];

   for (int i = 0; i < MAX; i++) {
      ptr[i] = &var[i];  // assign the address of integer.
   }

   for (int i = 0; i < MAX; i++) {
      cout << "Value of var[" << i << "] = ";
      cout << *ptr[i] << endl;
   }

   return 0;
}
```

When the above code is compiled and executed

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

DYNAMIC MEMORY

- Memory required may depends on problem size
- Memory required may not know at compile-time

STATIC MEMORY

- Compiler determines the amount of static memory (for each translation unit)
- Linker determines the static memory for entire application
- Fast, fixed, shared between variables and objects

STATIC MEMORY-LIFETIME

```
// lifetime of a local variable or object

for (int i = 0; i < 10; i++) {
double x = 0; // lifetime of x starts here
// ...
} // lifetime of x ends here

for (int i = 0; i < 10; i++) {
double y = 4; // lifetime of y starts here
// ...
} // lifetime of y ends here
```

DYNAMIC MEMORY

- During execution the application may request memory from OS
- Dynamic memory is allocated at run-time
- new and delete to allocate and de-allocate dynamic memory
- allocated memory must be de-allocated within the scope of the pointer that holds its address

NEW

- The new operation returns a pointer to the memory allocated
- Dynamic allocation of arrays,

```
pointer = new Type[size];
pointer = new Type; (single instance)
```

```
int n;                  // the number of students
Student* student = nullptr; // the address of the dynamic array

cout << "How many students in this section? ";
cin >> n;

student = new Student[n];   // allocates dynamic memory
```

DELETE

- Must deallocate the memory within the scope
- Delete [] pointer;
- delete pointer; (single instance)

```
delete [] student;
 student = nullptr;  // optional
```

A COMPLETE EXAMPLE

```
// Dynamic Memory Allocation
 // dynamic.cpp

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

 struct Student {
    int no;
    float grade[2];
 };

 int main( ) {
    int n;
    Student* student = nullptr;

    cout << "Enter the number of students : ";
    cin >> n;
    student = new Student[n];

    for (int i = 0; i < n; i++) {
        cout << "Student Number: ";
        cin  >> student[i].no;
        cout << "Student Grade 1: ";
        cin  >> student[i].grade[0];
        cout << "Student Grade 2: ";
        cin  >> student[i].grade[1];
    }

    for (int i = 0; i < n; i++) {
        cout << student[i].no << ": "
         << student[i].grade[0] << ", " <<
student[i].grade[1]
         << endl;
    }

    delete [] student;
    student = nullptr;
 }
```

MEMORY ISSUES

- Memory Leaks (loose the address before de-allocate)
  - Pointer out of scope
  - Pointer memory value changes
- Insufficient memory (will throw exception and stop execution)

QUIZ TWO

- Types, References and Overloading
- Dynamic Memory
- Member Functions and Privacy

NEXT WORKSHOP

- Compiling modules
- Dynamic Memory

QUESTIONS?

Thank You