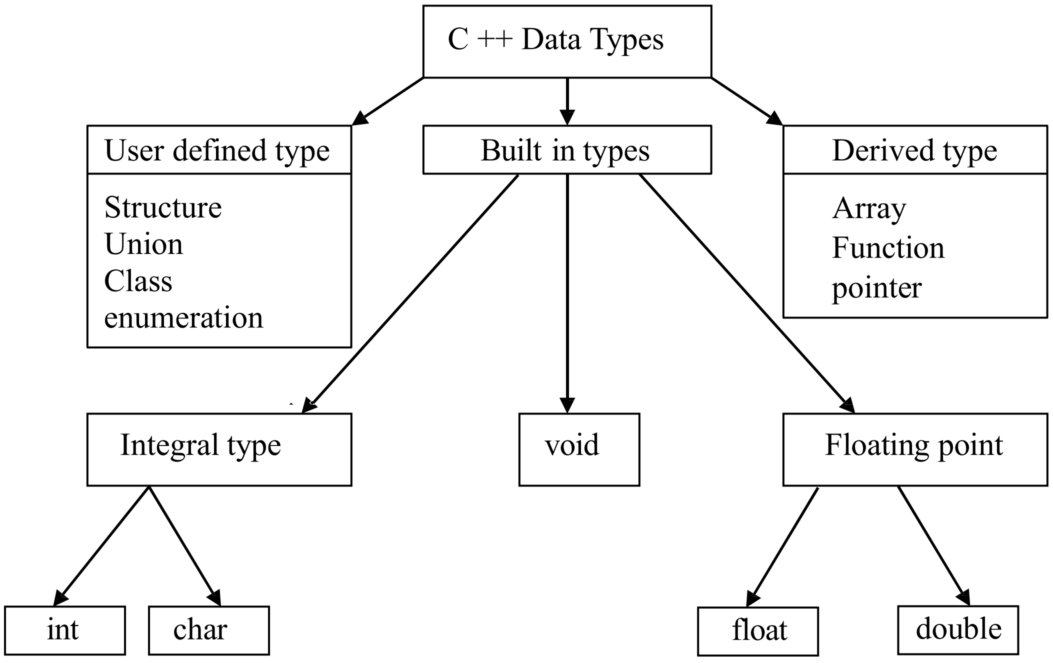
Lecture-3

Basic Data Types in C++

Both C and C++ compilers support all the built-in types. With the exception of void the basic datatypes may have several modifiers preceding them to serve the needs of various situations. The modifiers signed, unsigned, long and short may applied to character and integer basic data types. However, the modifier long may also be applied to double.



The type void normally used for:

In C++, the void keyword is used to represent the absence of a value. It is often used in various contexts, such as function return types, function parameters, and pointers.

- Void Functions

One of the most common uses of void is to declare functions that do not return a value. These functions are used for performing actions or tasks without producing a result. In this example, the greet function does not return any value; it simply prints a greeting message to the console.

```
#include <iostream>
void greet() {
    std::cout << "Hello, World!" << std::endl;
}
int main() {
    greet(); // Call the void function
    return 0;
}
```

- Void Pointers (void)*

Void pointers, denoted as void*, are used when you want to create a pointer that can point to data of any data type. You can later cast the void pointer to the appropriate type to access the data.

```
int number = 42;
float pi = 3.14159;
void* genericPtr; // Declare a void pointer

genericPtr = &number; // Store the address of an int
cout << "Value pointed to by genericPtr: " <<
*static_cast<int*>(genericPtr) <<endl;
genericPtr = &pi; // Store the address of a float
cout << "Value pointed to by genericPtr: "
<<*static_cast<float*>(genericPtr) <<endl;
```

- Void as a Placeholder

In some cases, void can be used as a placeholder in function prototypes or data structures when you want to specify that there's no specific return value or data type, or that it will be determined later.

```
void someFunction(int); // A function prototype indicating an unspecified
return type
typedef void (*FunctionPointer)(); // A typedef for a function pointer
with no return type
```

In this example, someFunction is declared with an unspecified return type, and FunctionPointer is a typedef for a function pointer that points to a function with no return type.

User Defined Data Types

We have used user defined data types such as struct, and union in C. While these more features have been added to make them suitable for object-oriented programming. C++ also permits us to define another user defined data type known as class which can be used just like any other basic data type to declare a variable. The class variables are known as objects, which are the central focus of oops.

Struct

In C++, a struct is a user-defined data type that allows you to group together variables of different data types under a single name. It is similar to a class, but with default public access for its members. structs are often used to represent a collection of related data fields. To create a struct, you use the struct keyword followed by the name of the struct and a set of variables enclosed in curly braces. Each variable is called a member of the struct.

```
struct Person {
    string name;
    int age;
};
```

Create instances (objects) of the struct and access its members using the dot (.) operator.

```
int main() {
    Person person1;
    person1.name = “Alice”;
    person1.age = 30;
    // Accessing struct members
    cout<< person1.name <<” is “<< person1.age << “ years old.” <<endl;
    return 0;
}
```

We can pass structs as function arguments to manipulate or display their data.

```
void displayPerson(const Person& p) {
    cout <<p.name<< " is " << p.age <<" years old."<<endl;
}
int main() {
    Person person1;
    person1.name = "Alice";
    person1.age = 30;
    displayPerson(person1);
    return 0;
}
```

We can also initialize a struct when declaring it, like this:

```
Person person1 = {"Alice", 30};
```

Enumerated Data Type

An enumerated data type is another user defined type which provides a way for attaching names to number, these by increasing comprehensibility of the code. The enum keyword automatically enumerates a list of words by assigning them values 0,1,2 and so on. This facility provides an alternative means for creating symbolic. Example:

```
enum shape { circle,square,triangle}
enum colour{red,blue,green,yellow}
enum position {off,on}
```

The enumerated data types differ slightly in C++ when compared with ANSI C. In C++, the tag names shape, colour, and position become new type names. That means we can declare new variables using the tag names. Example:

```
Shape ellipse;//ellipse is of type shape
colour background; // back ground is of type colour
```

ANSI C defines the types of enums to be ints. In C++, each enumerated data type retains its own separate type. This means that C++ does not allow an int value to be automatically converted to an enum.

Example:

```
colour background =blue; //vaid
colour background =7; //error in c++
colour background =(colour) 7;//ok
```

How ever an enumerated value can be used in place of an int value. Example:

```
int c=red ;//valid, colour type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second and so on. We can also write

```
enum color {red, blue=4,green=8};
enum color {red=5,blue,green};
```

C++ also permits the creation of anonymous enums (i.e, enums without tag names) Example:

```
enum{off,on};
```

Here off is 0 and on is 1. These constants may be referenced in the same manner as regular constants. Example:

```
int switch-1=off;
int switch-2=on;
```

ANSI C permits an enum defined with in a structure or a class, but the enum is globally visible. In C++ an enum defined with in a class is local to that class.

```
#include<iostream>
enum Color {
    RED,    // 0
    GREEN,  // 1
    BLUE    // 2
};
main()
{
       Color myFavoriteColor = GREEN;
       switch (myFavoriteColor) {
    case RED:
        std::cout << "You like red." << std::endl;
        break;
    case GREEN:
        std::cout << "You like green." << std::endl;
        break;
    case BLUE:
        std::cout << "You like blue." << std::endl;
        break;
    default:
        std::cout << "You have no favorite color." << std::endl;
}
```

In C++11 and later, you can create scoped enumerations (also known as "strong" or "enum class"). These provide better scoping and avoid polluting the global namespace. For example:

```
enum class Status {
    OK,
    ERROR
};
Status currentStatus = Status::OK;
```

Scoped enumerations are a good choice when you want to avoid naming conflicts and create more robust code. Enumerations are a useful feature in C++ for creating code that is easier to read, understand, and maintain by providing meaningful names to integral constants.

Union

In C++, a union is a user-defined data type that allows you to create a structure that can hold different data types but uses the same memory location. Unlike a struct, where all members have their memory allocated simultaneously, in a union, only one member is stored at any given time. Unions are useful when you need to represent data that can be of different types, but you want to save memory by ensuring that only one type's data is stored at a time. To create a union, you use the union keyword followed by the name of the union and a set of variables enclosed in curly braces. Each variable is called a member of the union. In this example, we've defined a union called Data with three members: an integer, a double, and a character.

```
union Data {
    int integer;
    double floating;
    char character;
};
```

You can create instances of the union and access its members using the dot (.) operator. However, only one member can be used at a time. In this example, we've created a Data union and stored an integer value in the integer member.

```
Data data;
data.integer = 42;
// Accessing the integer member
cout << "Integer: " << data.integer <<endl;
```

Unions save memory by sharing the same memory location for all members. When you assign a value to one member, it may overwrite the data of the other members. assigned a double value to the floating member, which overwrites the integer value stored previously.

```
data.floating = 3.14;
// Accessing the double member, but the integer value is lost
cout << "Double: " << data.floating <<endl;
```

Unions are useful when you need to represent data that can be of different types, but only one type's data is relevant at a given time. Common use cases include implementing variant data types or working with hardware registers that can store different types of data.

In C++11 and later, you can create an anonymously typed union as a member of a struct or class, providing a way to access different types of data through a shared memory location. In this example, the union is anonymously typed within the struct UnionHolder.

Example:

```
struct UnionHolder {
    union {
        int x;
        double y;
        char z;
    };
};
```

Be cautious when using unions, as they can lead to data type-related bugs if not used carefully. The responsibility of keeping track of which member is currently in use lies with the programmer. Unions in C++ provide a way to efficiently use memory to store different types of data within a single memory location. They are a valuable tool in specific situations where memory optimization and flexibility are required.

```
#include <iostream>
Using namespace std;
union Data {
    int integer;
    double floating;
    char character;
};
int main() {
    Data data;
    data.integer = 42;
    cout << "Integer: " << data.integer << endl;
    data.floating = 3.14;
    cout << "Double: " << data.floating << endl;
    data.character = 'A';
    cout << "Character: " << data.character << endl;
    // Note: When you access a union member, you may see unpredictable
results because
    //only the last assigned member has a valid value. In this case, 'A'
is printed
    // because it was the last assignment.
    return 0;
}
```

It's important to note that the choice between class, struct, enum, and union should depend on your specific requirements and design goals in your C++ program. The following Table comparison between class, struct, enum, and union in C++.

Feature	class	struct	enum	union
Definition	User-defined data type with private access control.	User-defined data type with public access control.	User-defined data type for named constants	User-defined data type with multiple members sharing same memory location.
Access Control	Private by default	Public by default		
Initialization	Can have constructors to initialize object.	Can have constructors to initialize object.	Enum values are implicitly assigned	Members can be initialized individually.
Member Variables	Members can have data types and functions.	Members can have data types and functions.	-	Members can have different data types.
Data Sharing	Objects created from classes can have private data	Objects created from structs have public data members.	Enum values are constants with integral values.	Share the same memory location for different data types.
Size and Memory	Generally larger due to additional member functions	Generally larger due to additional member functions	Minimal space (usually 4 bytes).	Size is determined by the largest member within the union.
Use Cases	Typically used to model objects and encapsulate data and behavior.	Often used for lightweight data structures and data members.	Creating symbolic names for integral constants.	Storing different types of data in a shared memory location.