Data Types

- Fundamental (built-in / primary / primitive) data types
- User-defined data types
- Compound (derived) data types

- Modern C++ features: auto and nullptr

Data Types

- **Fundamental (built-in / primary / primitive) data types**
- User-defined data types
- Compound (derived) data types

- Modern C++ features: auto and nullptr

Fundamental Data Types

- Like C language
  - char, int, float, double, void
  - Modifiers can be applied
    - signed and unsigned - int and char
    - short - int
    - long – int and double
    - long long - int
  - size and range of different data types depend on compiler and machine architecture

- C++ is stricter than C when it comes to data types
  - E.g. C++ does not treat character constants as integers
    - sizeof(char) and sizeof(‘A’) is always 1 in C++ according to standard
    - In C, sizeof(‘A’) is same as sizeof(int)
  - This behaviour is required for overload resolution (more on this later)

Fundamental Data Types

- New fundamental data types added to C++
  - **bool**
  - **wchar_t**, **char16_t** and **char32_t**

Fundamental Data Types

- **bool**
  - *bool* variable can only hold **true** or **false**
    - bool b1 = true;
    - bool b2 = false;
  - **bool**, **true** and **false** are three new **keywords**

  - *bool* variables and *true/false* keywords can be used in mathematical expression (e.g. **10 + b1 – false**)
    - In mathematical expression bool is elevated to int
      - *true* becomes 1 and *false* becomes 0
      - On assigning 0 to bool variable, it becomes *false;* on assigning any other value, it becomes *true*
    - Guess the output: **bool b = 2.5; int i = b; cout << i;**
    - Guess the output: **bool b = 2.5; cout << b**;

  - Default value of bool variables depends on storage class
    - Static and global variables are *false* by default
    - Non-static local variables may be *true*/*false* by default
  - Size of bool variables is implementation dependent

Fundamental Data Types

- **wchar_t**, **char16_t** and **char32_t**
  - You will rarely encounter these data types
  - We will not explore these

Data Types

- Fundamental (built-in / primary / primitive) data types
- **User-defined data types**
- Compound (derived) data types

- Modern C++ features: auto and nullptr

User-defined Data Types

- *struct* and *union*
  - Can be used as in C language
  - Many new features have been added for OOP
  - More about new features when we start OOP

- New data type called *class* have been added
  - Major addition to C++ to enable OOP
  - More about classes when we start OOP

- Enumerated data types (enum)
  - You can use enum like C language
  - There are new features added in C++ related to enum
    - We will not explore them

Data Types

- Fundamental (built-in / primary / primitive) data types
- User-defined data types
- **Compound (derived) data types**
  - **Arrays**
  - Pointers
  - References

- Modern C++ features: auto and nullptr

Arrays

- Similar to C language
- One exception is related to initialization of character array
  - char name[3] = “RAM”; //Valid in C, **Error in C++**
    - In C++, you must count for ending null character during static initialization of char array

Data Types

- Fundamental (built-in / primary / primitive) data types
- User-defined data types
- **Compound (derived) data types**
  - Arrays
  - **Pointers**
  - References

- Modern C++ features: auto and nullptr

Pointers

- Same as C language
- Let us revise pointer to constant Vs Constant pointer

```
char name1[5] = "DDU";
char name2[5] = "JNU";
char *const ptr1 = name1;         // ptr1[0] = 'P';  ~~ptr1 = name2;~~
char const *ptr2 = name1;         // ~~ptr2[0] = 'P';~~  ptr2 = name2;
const char *ptr3 = name1;         // ~~ptr3[0] = 'P';~~  ptr3 = name2;
char const *const ptr4 = name1;   // ~~ptr4[0] = 'P';~~ ~~ptr4 = name2;~~
```

Pointers

- Unlike C language, in C++ void pointer can't be assigned to non-void pointer without type cast. (**Why?**)

```
void *vptr; char *cptr;
cptr = vptr                  // valid in C, not in C++
cptr = (char *)vptr;         // valid in C and C++
vptr = cptr;                 // valid in C and C++
```

Data Types

- Fundamental (built-in / primary / primitive) data types
- User-defined data types
- **Compound (derived) data types**
  - Arrays
  - Pointers
  - **References**

- Modern C++ features: auto and nullptr

References: What is reference?

- New concept in C++
- Creates an **alias** (alternate name) for a variable
- Declaration is as follows:
  data-type &reference-name = variable-name;
  int mumbai = 10;
  int &bombay = mumbai;

- **mumbai** and **bombay** are referring to the **same memory location**, changing one will change other
- Please note that neither mumbai nor bombay is pointer here
- mumbai and bombay can be used interchangeably

References: Initialization

- Reference variable **must be initialized** at the time of declaration
  - Initialization of reference variable is completely different from initialization of non-reference variable

```
int mumbai = 10;
int chennai = mumbai;           // Initialization of int
int &bombay = mumbai;           // Initialization of int ref
bombay = chennai;
cout << mumbai;
```

- Unlike pointer, **reference can not refer to nothing**
  - Pointer can be a NULL pointer (pointing to nothing)
  - Reference must refer to something
- Unlike pointer, **once initialized, reference variable can not be changed to refer to any other variable**

References: no chaining, no array of ref

- Reference variable can be initialized with other reference variables of same type, but its not chaining like pointers

```
int i;
int &ir1 = i;
int &ir2 = ir1;
```

  - Addresses of *i*, *ir1* and *ir2* would be same in above case. And **both ir1 and ir2 are int references**
  - There **can not be a reference to reference** (no chaining)

- **Array of references can not be created**
- But references can be created for arrays

```
int arr[10] = {1, 2};
int &ref = arr[5];          // ref is reference to int (one array element - arr[5])
int (&arr2)[10] = arr;      // arr2 is reference to whole array arr
cout << arr2[1] << arr2[3]; // Guess the output
```

References: no pointer to ref

- Pointer to reference is not allowed

```
int &ir = i;
// error: cannot declare pointer to 'int&'
// int &*irp = &ir;
```

- Reference to pointer is allowed.

```
int i = 7;
int *ip = &i;
int *&ipr = ip;      // ipr is reference to int *
cout << *ipr;        // Prints 7
```

- For more information refer **this** link

- It is possible to create reference to function
    - we will not look into it

References: clarifications

- Clarification I:
```
int i = 7;
int *ip = &i;  // ip is just int pointer
int &ir = *ip; // ir is a int reference, not reference to pointer
cout << ir;
```

- Clarification II:
```
int i;
int &ir = i;
int *ip = &ir; // ip is just an int pointer
```

References: constant reference

```
int i = 1;
const int ci = 2;
int &ref1 = i;              // Valid
// int &ref2 = ci;          // Invalid
const int &cref1 = i;       // Valid
const int &cref2 = ci;      // Valid
ref1= 5;                    // Valid
// cref1 = 5;               // Invalid
// cref2 = 5;               // Invalid
```

- As seen earlier, **constant pointer** and **pointer to const** are **separate concepts**
  - We can not change constant pointer to point to some other variable (than what it has been initialized to point)
  - Using pointer to const we can not alter value that is stored at location being pointed by it

- References themselves are always constant by nature
  - So we will use **constant reference** and **reference to constant** interchangeably
  - Using **constant reference** we can not alter value being referred

References: reference to temporary and user-defined types

- **References can be created for temporary objects** like literal constants, sum of two variables, return value of function etc.
  - References to temporaries must be declared as constant

    const char &ref1 = ‘A’;
    const int &ref2 = i + j; // where i and j are int variables
    const float &ref3 = fun(); // where fun() returns float value

- **Lifetime of temporary objects is tied to lifetime of its reference**

- References can be created for user-defined data types too

```
struct s {
    int i;
} s1;
struct s &sr = s1;
```

(Refer Annotations Page-1,2)

References to const

Examples:

```
const int ci = 1024;
const int &r1 = ci;   // ok: both reference and underlying object
                            are const
r1 = 42;              // error: r1 is a reference to const
int &r2 = ci;         // error: non const reference to a const object
```

Initialization and References to const

We can bind a reference to const to a nonconst object, a literal, or a more general expression.

Examples:

```
int i = 42;
const int &r1 = i;          // we can bind a const int& to a plain
                               int object
const int &r2 = 42;         // ok: r2 is a reference to const
const int &r3 = r1 * 2;     // ok: r3 is a reference to const
int &r4 = r1 * 2;           // error: r4 is a plain, non const
                               reference
```

Initialization and References to const

The same initializations are not legal for nonconst references.

They result in compile-time errors.

But why do they work for const references ?

Let's try to understand this behavior by taking a look at what happens when we bind a reference to an object of a different type.

If we write:

```
double dval = 3.14;
const int &ri = dval;
```

The compiler transforms this code to

```
int temp = dval;        //create temporary
                          int from the double
const int &ri = temp; // bind ri to that
                          temporary
```

References: call by reference

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

void fun(int &num) {
    num++;
}

int main() {
    int i = 10;
    fun(i);
    cout << i;

    return 0;
}
```

Output:
11

- When a function receives argument by reference and changes, any changes in its value will be reflected in calling function
- Here variable *num* in function *fun* is reference to variable *i* in *main* function. Changing *num* in *fun* will change *i* in *main*.

References: return by value

```
int fun(int &num) {
    num++;
    return num;
}

int main() {
    int i = 10;
    // ret_val is ref to temporary object
    const int &ret_val = fun(i);
    cout << i << " " << ret_val << endl;
    cout << &i << " " << &ret_val << endl;

    return 0;
}
```

Output:
11 11
0x7fff247b9b68 0x7fff247b9b6c

- When a function returns by value, a temporary copy is returned to the calling function.
- Calling function then can copy that temporary to another variable or can use reference to refer to that temporary.

References: return by value

```
int fun(int &num) {
    num++;
    return num + 1;
}

int main() {
    int i = 10;
    // ret_val is ref to temporary object
    const int &ret_val = fun(i);
    cout << i << " " << ret_val << endl;
    cout << &i << " " << &ret_val << endl;

    return 0;
}
```

Output:
11 12
0x7fff247b9b68 0x7fff247b9b6c

- Here *fun* creates temporary object for storing result of *num + 1*
- As *fun* is returning by value, it sends another copy of that temporary to the *main* function
- *ret_val* refers to temporary copy present in activation record of *main* and not to the temporary copy present in activation record of *fun*

References: return by reference

(Refer Annotations Page-3,4)

```
int &fun(int &num) {
    cout << num << endl;
    num++;
    return num;
}

int main() {
    int i = 10;
    int res = fun(i);
    int &ret_val = fun(i);
    fun(i) += 2;
    cout << i << " " << res << " " << ret_val << endl;
    cout << &i << " " << &res << " " << &ret_val << endl;
    return 0;
}
```

```
10
11
12
15 11 15
0x7fff78e63b28 0x7fff78e63b2c 0x7fff78e63b28
```

- When function returns a reference, instead of temporary copy, calling function can use reference return by called function, to copy value to local variable or create another reference to it
- If function is returning a reference then function call can be on the left hand side of the assignment operator

References: return by reference

```
const int &fun(int &num) {
    cout << num << endl;
    num++;
    // returning reference to temporary
    return num + 1;
}

int main() {
    int i = 10;
    const int &ret_val = fun(i);
    cout << i << " " << " " << ret_val << endl;
    cout << &i << " " << " " << &ret_val << endl;
    return 0;
}
```

- If temporary is being returned by the function then function return type must be constant, otherwise compiler will generate an error.
- Now as we have used const reference as return type the code will compile
- But once we leave *fun* function temporary will get destroyed. So there is no point of returning a reference to temporary. **It will result in undefined behaviour (may be segmentation fault).**

- While returning by reference, developer must pay attention to scope of the variable/temporary whose reference is being returned.
- Never return variable or temporary whose lifetime ends with end of function execution
- This code results in segmentation fault as temporary of fun is accessed in main (using ret_val)

References: return by reference

```
const int &fun(int &num) {
    cout << num << endl;
    num++;
    // returning reference to temporary, hence invalid code
    return num++;
}

int main() {
    int i = 10;
    const int &ret_val = fun(i);
    cout << i << " " << " " << ret_val << endl;
    cout << &i << " " << " " << &ret_val << endl;
    return 0;
}
```

- It is returning temporary as it needs to increment *num* and then return previous value of *num*.
- So it creates a temporary copy of *num* before incrementing and returns reference to that temporary copy

(Refer Annotations Page-5, 6)

References: return by reference

```
const int &fun(int &num) {
    cout << num << endl;
    num++;

    return ++num;
}

int main() {
    int i = 10;
    const int &ret_val = fun(i);
    cout << i << " " << " " << ret_val << endl;
    cout << &i << " " << " " << &ret_val << endl;
    return 0;
}
```

10
12 12
0x7fff8d1eb01c 0x7fff8d1eb01c

- Function *fun* returns reference to variable *num* which is nothing but reference to variable *i* in *main* function itself
- Return type of function *fun* and variable *ret_val* in *main* does not have to be constant

References: return by reference

```
const int &fun(int &num) {
    cout << num << endl;
    num++;

    return num += 1;
}

int main() {
    int i = 10;
    const int &ret_val = fun(i);
    cout << i << " " << " " << ret_val << endl;
    cout << &i << " " << " " << &ret_val << endl;
    return 0;
}
```

10
12 12
0x7fff8d1eb01c 0x7fff8d1eb01c

- Function *fun* returns reference to variable *num* which is nothing but reference to variable *i* in *main* function itself
- Return type of function *fun* and variable *ret_val* in *main* does not have to be constant

References Vs Pointers

- **References are limited in capability compared to pointers**
  - But that makes references **easy to use and simple to understand compared to pointers**
    - Don't need to worry about NULL references
    - Can’t be changed to refer to other variables once declared
    - No arrays of references
    - No chaining (No reference to reference)
    - No pointers to references
  - Anything that can be done using references can be achieved using pointers
- **Use references when possible**, use pointers when it is must
- Internally, most compilers implement references using pointers

Data Types

- Fundamental (built-in / primary / primitive) data types
- User-defined data types
- Compound (derived) data types

- **Modern C++ features**: ***auto*** and *nullptr*

auto

- *auto* keyword is present in C language and C++98, but its meaning is very different. And it is almost never used
- Meaning of *auto* has changed in C++11
- Since C++11, *auto* keyword can be used to declare a variable; and it's **type will be derived from type of data used to initialize variable**
- Hence it is **mandatory to initialize variable** when it is declared using *auto*

```
auto i = 3;        // type of i is int as literal 3 is of type int
auto d = 4.5;      // type of d is double as 4.5 is of double type
auto &ir = i;      // ir is int reference as it has been declared as ref
auto &dr = d;      // dr is double reference
auto x;            // Invalid
```

auto

- *auto* is very useful when type name is very long. We will encounter such cases going forward
  - Simple example

```
int arr[10] = {1, 2};
int *const &arr2 = arr;       // arr2 is reference to constant int pointer
int (&arr3)[10] = arr;        // arr3 is reference to entire array arr
auto &arr4 = arr;             // arr4 is reference to entire array arr

cout << arr[0] << " " << arr[2] << endl;       // Prints 1 and 0
cout << arr2[0] << " " << arr2[2] << endl;     // Prints 1 and 0
cout << arr3[0] << " " << arr3[2] << endl;     // Prints 1 and 0
cout << arr4[0] << " " << arr4[2] << endl;     // Prints 1 and 0

cout << sizeof(arr) << endl;       // 40
cout << sizeof(arr2) << endl;      // 8
cout << sizeof(arr3) << endl;      // 40
cout << sizeof(arr3) << endl;      // 40
```

Data Types

- Fundamental (built-in / primary / primitive) data types
- User-defined data types
- Compound (derived) data types

- **Modern C++ features**: *auto* and ***nullptr***

nullptr

- ***nullptr* should be used instead of NULL**

```
int *ip = nullptr;        // Prefer this
~~int *ip = NULL;~~       // Avoid this
char *cp = nullptr;       // Prefer this
~~char *cp = NULL;~~      // Avoid this
```

- This **helps in overload resolution** as *nullptr* has type *std::nullptr_t*, while NULL is integer constant zero
  - More about this later when we learn function overload resolution
- When you intialize variable with *nullptr* then use *nullptr* for comparision too
  - if(**ip == nullptr**)

Questions - I

- What will be the output of the following code?

  for(auto i = **3u**; **i >= 0**; i--)

  std::cout << i << " ";

- What will be the output of the following code?

  for(auto i = **3u**; **i > 0**; i--)

  std::cout << i << " ";

Questions - II

- What will be the output of the following code?
```
auto f = 3.14;
double &r = f;
f = 4.14;
std::cout << f << " " << r;
```

- What will be the output of the following code?
```
auto f = 3.14;
const int &r = f;
f = 4.14;
std::cout << f << " " << r;
```

- What will be the output of the following code?
```
auto f = 3.14;
const double &r = f;
f = 4.14;
std::cout << f << " " << r;
```

- What will be the output of the following code?
```
auto f = 3.14;
int &r = f;
f = 4.14;
std::cout << f << " " << r;
```