Data Types in C++:

A **data type** defines the type of data a variable can store. It determines the size, layout in memory, and the operations that can be performed on the data.

Categories of Data Types

C++ data types are divided into three main categories:

1. **Primitive Data Types (Built-in)**: Basic types provided by C++.
2. **Derived Data Types**: Derived from fundamental types (e.g., arrays, pointers, references).
3. **User-Defined Data Types**: Created by users (e.g., structures, classes, enums).

Primitive Data Types

Data Type	Description	Size (bytes)	Example
int	Integer numbers	4 (typically)	int x = 10;
float	Single-precision decimals	4	float pi = 3.14f;
double	Double-precision decimals	8	double e = 2.71;
char	Single character	1	char ch = 'A';
bool	Boolean (true/false)	1	bool flag = true;
void	No data (used for functions)	0	void func();
wchar_t	Wide character	2 or 4	wchar_t w = L'A';

---

1. Integer (int)

- Stores whole numbers, positive or negative.
- Cannot store decimals.

Example:

int a = 5, b = -10;

cout << "a: " << a << ", b: " << b << endl;

Output:

a: 5, b: -10

---

2. Floating-Point (float and double)

- Stores numbers with decimals.
- Use float for less precision, double for more precision.

Example:

float f = 3.14f;

double d = 2.718281828459;

cout << "Float: " << f << ", Double: " << d << endl;

Output:

Float: 3.14, Double: 2.71828

---

3. Character (char)

- Stores a single character in single quotes ('A').
- Internally, stored as an ASCII value.

Example:

char letter = 'A';

cout << "Character: " << letter << endl;

Output:

Character: A

---

4. Boolean (bool)

- Stores true (1) or false (0).

Example:

bool isHappy = true;

cout << "Is Happy: " << isHappy << endl;

Output:

Is Happy: 1

---

5. Wide Character (wchar_t)

- Stores characters larger than a char, often used for Unicode.

Example:

```
wchar_t w = L'A';
wcout << L"Wide Character: " << w << endl;
```

Derived Data Types

Derived Type	Description	Example
Array	Collection of similar elements	int arr[5];
Pointer	Stores the address of a variable	int *ptr;
Reference	Alias for another variable	int &ref = x;

Example (Array):

```
int arr[3] = {1, 2, 3};
cout << "First Element: " << arr[0] << endl;
```

User-Defined Data Types

User-Defined Type	Description	Example
struct	Collection of variables	struct Student {...}
class	Blueprint for objects	class Car {...}
enum	Enumerated constants	enum Color {...}
typedef/using	Alias for a data type	typedef int Age;

Example (Structure):

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

```
Student s1 = {"John", 20};

cout << "Name: " << s1.name << ", Age: " << s1.age << endl;
```

Modifiers for Data Types

Modifiers alter the size or behavior of data types.
Common modifiers:

- **signed**: Default for int (positive and negative numbers).
- **unsigned**: Only positive numbers.
- **short**: Uses less memory.
- **long**: Uses more memory.

Type	Size (bytes)	Range
signed int	4	-2,147,483,648 to 2,147,483,647
unsigned int	4	0 to 4,294,967,295
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
long int	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
long double	16	Depends on system

Examples of Modifiers

Unsigned Integer

```
unsigned int count = 10;

cout << "Count: " << count << endl;
```

Output:

```
Count: 10
```

Long Integer

```
long int bigNumber = 123456789;

cout << "Big Number: " << bigNumber << endl;
```

Output:

```
Big Number: 123456789
```

Type Conversion

1. Implicit Conversion

- Automatic conversion by the compiler.

Example:

```
int x = 10;
double y = x; // int to double
cout << "y: " << y << endl;
```

Output:

```
y: 10.0
```

2. Explicit Conversion (Type Casting)

- Manually convert data types using casting.

Example:

```
double pi = 3.14159;
int truncatedPi = (int)pi; // Cast double to int
cout << "Truncated Pi: " << truncatedPi << endl;
```

Output:

```
Truncated Pi: 3
```

Key Points

1. Use appropriate data types based on the range and precision needed.
2. Understand the memory size and range of each data type.
3. Use modifiers for optimizing memory usage when needed.
4. Be cautious with type conversions to avoid data loss or unexpected behavior.