# CS135; User-Defined Simple Data Types, Namespaces, and the `string` Type

Gael Zarco

April 8, 2025

C++ simple data types are split intro three categories:

1. Integral
2. Floating-point
3. `enum`

Listing 1: `namespace` Example

```cpp
using namespace std;
```

The above is used in every C++ program that follows the ANSI/ISO Standard C++ header files.

## 1 Enumerations

Data-types allow you to specify what values are legal and tell the user what kinds of operations are allowed on those values. Enumerations allow you to create simple data types.

The **Enumeration Type** requires the following:

1. A name for the data type.
2. A set of values for the data type.
3. A set of operations on the values.

Listing 2: Enumeration Syntax

```cpp
enum typeName {value1, value2, ...};
```

- `value1, value2, ...` are indentifiers called **enumerators**.
- `enum` is a reserved word in C++.

- The values between the braces can have a specified order between them.
  - This mean the enumeration type is an ordered set of values.
  - The default value assigned to these enumerators starts at 0. Increments for every value thereafter.
- The enumerators are **NOT** variables.

### 1.1 Declaring Variables

Once a data type is defined, you can declare variables of that type.

Listing 3: `enum` Variable Declaration Syntax

```
dataType identifier, identifier, ...;

\\ Example
enum sports {BASKETBALL, FOOTBALL, SOCCER, BASEBALL, VOLLEYBALL};
```

The following statement declares `popularSport` and `mySport` to be of type `sports`.

### 1.2 Assignment

Once the var is declared, you can store values in it.

Listing 4: `enum` Variable Assignment

```
popularSport = FOOTBALL;

// Copy the value of popularSport to mySport
mySport = popularSport;
```

### 1.3 Operations on Enumeration Types

No arithmatic operations are allowed on the enumeration type. This includes the increment and decrement operator. Use a cast when necessary:

Listing 5: `enum` Cast Example

```
popularSport = static_cast<sports>(popularSport + 1);
```

This statement advances the the value of `popularSport` to the next value in the list.

### 1.4 Relational Operators

Because an enumeration is ordered, relational operators can be used.

Listing 6: Enumeration Type Relational Operator Example

```
FOOTBALL <= SOCCER    \\ true
```

## 1.5 Enumeration Types and Loops

Enumeration type is an integral type and can be used to increment, decrement, and compare the values of the enumeration type (using cast).

Listing 7: Enumeration Type Loop

```
for (mySport = BASKETBALL; mySport <= VOLLEYBALL; mySport =
  static_cast<sports>(mySport + 1))
```

## 1.6 Input/Output of Enumeration Types

The enumeration type cannot be used for neither input or output (directly).

## 1.7 Functions and Enumeration Types

The enumeration type can be passed as a param to funcs by value or reference. Funcs can also return a value of the enum type.

Listing 8: `enum` Types in Functions

```
void printEnum(courses registered) {
  switch (registered) {
  case ALGEBRA:
  cout << "Algebra";
  break;
  case ANALYSIS:
  cout << "Analysis";
  break;
  case BASIC:
  cout << "Basic";
  break;
  case CHEMISTRY:
  cout << "Chemistry";
  break;
  case CPP:
  cout << "CPP";
  break;
  case HISTORY:
  cout << "History";
  break;
  case PYTHON:
  cout << "Python";
  break;
  case PHILOSOPHY:
  cout << "Philosophy";
  }    //end switch
}      //end printEnum
```

## 1.8 Declaring Variables When Defining the Enumeration Type

Listing 9: Simultaneous `enum` Declaration and Variable Declaration

```c
enum grades {A, B, C, D, F} courseGrade;
```

This listing defines an enumeration type, `grades` and declares a variable `courseGrade` of type `grades`.

## 1.9 Anonymous Data Types

A data type where you directly specify values in the variable declaration with no type name is called an **Anonymous Type**.

Listing 10: Anonymous Enumeration Type Example

```c
// No name is given to the data type
enum {BASKETBALL, FOOTBALL, BASEBALL, HOCKEY} mySport;
```

Drawbacks:

1. Anonymous data types cannot be passed as params to a func and a func can not return an anonymous data type.
2. Values used in one anon data type can be used in a separate anon data type, but variables of those types are treated differently.

Listing 11: illegal `enum` operations

```c
enum {english, french, spanish, german, russian} languages;
enum {english, french, spanish, german, russian} foreignlanguages;

languages = foreignlanguages;    // illegal
```

## 1.10 `typedef` Statement

You can create synonyms or aliases to a previously defined data type by using the `typedef` statement.

Listing 12: `typedef` Statement Syntax

```c
typedef existingTypeName newTypeName;
```

- `typedef` is a reserved word.
  - Does not create any new data type; only creates an alias to the existing type.

## 2 Namespaces

When a header file such as `iostream` is included in the program, the global identifiers in the header file also become global identifiers in the program.

- This means that the compiler may return a syntax error if a global identifier is redefined.
- Third-party libraries and other software usually add a special character to the beginning of their global identifiers to avoid compiler errors.
  - Do not do this in your own code.

ANSI/ISO Standard C++ tries to solve this problem of overlapping identifiers with the `namespace` mechanism.

Listing 13: `namespace` Statement Syntax

```
namespace namespace_name {
  members
}
```

Where `members` is usually named constants, variable declarations, functions or another `namespace`.

- `namespace` is a reserved word.

Listing 14: `namespace` Example

```
namespace globalType {
  const int N = 10;
  const double RATE = 7.50;
  int count = 0;
  void printResult();
}
```

This listing defined `globalType` to be a `namespace` with four members with their respective types.

- The scope of a `namespace` member is local to `namespace`.

To use a `namespace` member, you may:

1. Use `namespace::identifier` (scope resolution operator).
2. Use a `using` statement.

Listing 15: `using` Statement Syntax (Two Ways)

```
using namespace namespace_name;

using namespace::identifier;
```

## 3 string Type

The `string` type is a programmer-defined type and is not part of the C++ language; The C++ standard library supplies it. Therefore you must include the header file in your program.

Listing 16: `string` Type Example

```
string name = "Gael Zarco";
```

- Strings are sequences of zero or more characters, therefore each charcter has an assigned position starting with 0 (0, 1, 2, etc.).
- Strings can store just about any size string.
- The binary `+` operator and the array index (subscript) `[]` operator are defined operations for strings.

The **Array Subscript Operator** resembles `[]` and can be used to access a position of a character within a string.

Listing 17: Indexing Into A `string`

```
string str1 = "Hello there";

// Strings begin at index 0, 6 affects the 5th char
str1[6] = 'T';
```

### 3.1 Additional string Operations

- `string` has a data type `string::size_type`.
- `string` has a named constant `string::npos`.

| | |
|---|---|
| `string::size_type` | An unsigned integer (data) type |
| `string::npos` | The maximum value of the (data) type `string::size_type`, a number such as `4294967295` on many machines |

There are many other functions for string manipulation:

| Expression | Effect |
|---|---|
| `strVar.at(index)` | Returns the element at the position specified by `index`. |
| `strVar[index]` | Returns the element at the position specified by `index`. |
| `strVar.append(n, ch)` | Appends `n` copies of `ch` to `strVar`, where `ch` is a `char` variable or a `char` constant. |
| `strVar.append(str)` | Appends `str` to `strVar`. |
| `strVar.clear()` | Deletes all the characters in `strVar`. |
| `strVar.compare(str)` | Returns `1` if `strVar > str`; returns `0` if `strVar == str`; returns `-1` if `strVar < str`. |
| `strVar.empty()` | Returns `true` if `strVar` is empty; otherwise it returns `false`. |
| `strVar.erase()` | Deletes all the characters in `strVar`. |
| `strVar.erase(pos, n)` | Deletes `n` characters from `strVar` starting at position `pos`. |

| Expression | Effect |
|---|---|
| `strVar.find(str)` | Returns the index of the first occurrence of `str` in `strVar`. If `str` is not found, the special value `string::npos` is returned. |
| `strVar.find(str, pos);` | Returns the index of the first occurrence at or after `pos` where `str` is found in `strVar`. |
| `strVar.find_first_of(str, pos)` | Returns the index of the first occurrence of any character of `strVar` in `str`. The search starts at `pos`. |
| `strVar.find_first_not_of (str, pos)` | Returns the index of the first occurrence of any character of `str` not in `strVar`. The search starts at `pos`. |
| `strVar.insert(pos, n, ch);` | Inserts `n` occurrences of the character `ch` at index `pos` into `strVar`; `pos` and `n` are of type `string::size_type`; and `ch` is a character. |
| `strVar.insert(pos, str);` | Inserts all the characters of `str` at index `pos` into `strVar`. |
| `strVar.length()` | Returns a value of type `string::size_type` giving the number of characters in `strVar`. |
| `strVar.replace(pos, n, str);` | Starting at index `pos`, replaces the next `n` characters of `strVar` with all the characters of `str`. If `n >` length of `strVar`, then all the characters until the end of `strVar` are replaced. |
| `strVar.substr(pos, len)` | Returns a string which is a substring of `strVar` starting at `pos`. The length of the substring is at most `len` characters. If `len` is too large, it means "to the end" of the string in `strVar`. |
| `strVar.size()` | Returns a value of type `string::size_type` giving the number of characters in `strVar`. |
| `strVar.swap(str1);` | Swaps the contents of `strVar` and `str1`. `str1` is a `string` variable. |