CS302 - Data Structures
using C++

Pre-Course: Variables, Basic Types, Control Structures

Kostas Alexis

Slides inspired by the course "Modern C++", Uni Bonn: http://www.ipb.uni-bonn.de/teaching/modern-cpp/

Declaring Variables

- Variable declaration follows a certain pattern

  `<TYPE> <NAME> [ = <VALUE>];`

- Every variable has a type
- Variables cannot change their type
- Good practice: initialize variables if you can

```
int unitialized_var; // not good practice
bool initialized_var = true; // good practice
```

Naming Variables

- Name must start with a letter
- Give variables meaningful names
- Don't be afraid to use longer names
- Don't include type in the name
- Don't use negation in the name
- `GOOGLE-STYLE` name variables in snake_case all lowercase, underscores separate words
- C++ is case sensitive

Built-in Types

```
bool programming_is_great = true;       // Boolean: true or false
char carret_return = '\n';              // Single character
int your_favorite_number = 10;          // Integer number
short small_int = 10;                   // Short number
long big_int = 10;                      // Long number
float fraction = 0.01f;                 // Single precision float
double precise_num = 0.01;              // Double precision float
auto some_int = 13;                     // Automatic type [int]
auto some_float = 13.0f;                // Automatic type [float]
auto some_double = 13.0;                // Automatic type [double]
```

Operations on arithmetic types

- All character, integer and floating point types are arithmetic
- Arithmetic operations: `+`, `-`, `&`, `/`
- Comparisons `<`, `>`, `<=`, `>=`, `==` return bool
- `a += 1` → `a = a+1`, same for `-=`, `*=`, `/=`, etc
- Avoid `==` for floating point types

Some additional operations

- Boolean variables have logical operations: `||` (or), `&&` (and), `!` (not)

```
bool is_nice_temperature = (!is_cold && !is_hot) || is_around25C;
```

- Additional operations on integer variables:
  - `/` is integer division: `7/3 == 2`
  - `%` is module division: `7%3 == 1`
  - Increment operator: `a++` → `++a` → `a+=1`
  - Derement operator: `a--` → `--a` → `a -=1`
  - Do not use (de)increment operators within another expression (e.g., `a = (a++) + ++ b`)

Strings

- `#include <string>` to use `std::string`
- Concatenate strings with `+`
- Check if `str` is empty with `str.empty()`
- Works out of the box with I/O streams

```
#include <iostream>
#include <string>

int main()
{
     std::string help = "Hello";
     std::cout << "Type your name:" << std::endl;
     std::string name = "";   // Init empty
     std::cin >> name;        // Read name
     std::cout << hello + ", " + name + "!" << std::endl;
     return 0;
}
```

Use std::array for fixed size collection of items

- `#include <array>` to use `std::array`
- Store a collection of items of same type
- Create from data: `array<float, 3> arr = {1.0f, 2.0f, 3.0f};`
- Access items with `arr[i]` – indexing starts with 0
- Number of stored items: `arr.size()`
- Useful access aliases
  - First item: `arr.front() == arr[0]`
  - Last item: `arr.back() == arr[arr.size() - 1]`

Use std::vector when number of items is unknown before-wise

- `#include <vector>` to use `std::vector`
- Vector is implemented as a dynamic table
- Access stored items just like in `std::array`
- Remove all elements: `vec.clear()`
- Add a new item in one of two ways:
  - `vec.emplace_back(value)` `// preferred, c++11`
  - `vec.push_back(value)` `// historically better known`
- Consider it to be a default container to store collections of items on any same type

Optimize vector resizing

- Many **`push_back/emplace_back`** operations force vector to change its size many times
- **`reserve(n)`** ensures that the vector has enough memory to store **`n`** items
- The parameter n can een be approximate

```
std::vector<std::string> vec;
const int kIterNum = 100;
// Always call reserve if you know the size
vec.reserve(kIterNum);
for (int i=0; I < kIterNum; ++i)
{
     vec.emplace_back("hello");
}
```

Example vector

```
#include <string>
#include <vector>
#include <iostream>

using namespace std;
int main()
{
     vector<int> numbers = {1, 2, 3};
     vector<string> names = {“name1”, “name2”};
     names.emplace_back(“another_string”);
     cout << “First name: ” << names.front() << endl;
     cout << “Last number: ” << numbers.back() << endl;
     return 0;
}
```

Variables live in scopes

- There is a single global scope
- Local scopes start with { and end with }
- All variables belong to the scope where they have been declared
- All variables die in the end of their scope
- This is the core of C++ memory system

```
int main() {          // start of main scope
     float some_float = 13.13f;          // create variable
     {     // new inner scope
          auto another_float = some_float;          // copy variable
     }     // another_float dies
     return 0;
}          // some_float dies
```

Any variable can be const

- Use `const` to declare a constant
- The compiler will guard it from any changes
- Keyword `const` can be used with any type
- `GOOGLE-STYLE` names constants in `CamelCase` starting with a small letter `k`:
  - const float kImportantFloat = 20.0f;
  - const in kSomeInt = 20;
  - const std::string kHello = “hello”;
- const is part of type: variable `kSomeInt` has type `const int`
- Good practice: declare everything `const` unless it has to change

References to variables

- We can create a reference to any variable
- Use `&` to state that a variable is a reference
  - `float& ref = original_variable`
  - `std::string& hello_ref = hello;`
- Reference is part of type: variable ref has type `float&`
- Whatever happens to a reference happens to the variable and vice versa
- Yields performance gain as references avoid copying data.

References to variables

- We can create a reference to any variable
- Use `&` to state that a variable is a reference
  - `float& ref = original_variable`
  - `std::string& hello_ref = hello;`
- Reference is part of type: variable ref has type `float&`
- Whatever happens to a reference happens to the variable and vice versa
- Yields performance gain as references avoid copying data.

Const with references

- References are fast but reduce control
- To avoid unwanted changes use **const**
  - **const float&** ref = original_variable;
  - **const std::string&** hello_ref = hello;

```
#include <iostream>
using namespace std;
int main() {
     int num = 42;
     int& ref = num;
     const int& kRef = num;
     ref = 0;
     cout << ref << " " << num << " " << kRef << endl;
     num = 42;
     cout << ref << " " << num << " " << kRef = endl;
     return 0;
}
```

If statement

```
if (STATEMENT) {
          // this is executed if STATEMENT == true
} else if (OTHER_STATEMENT) {
          // this is executed if (STATEMENT == false) && (OTHER_STATEMENT == true)
} else {
          // this is executed if neither is true
}
```

- Used to conditionally execute code
- All the `else` cases can be omitted if needed
- `STATEMENT` can be any Boolean expression

Switch statement

```
switch (STATEMENT) {
     case CONST_1:
          // this is executed if STATEMENT == CONST_1
          break;
     case CONST_2:
          // this is executed if STATEMENT == CONST_2
          break;
     default:
          // this runs if no other option worked
}
```

- Used to conditionally execute code
- Can have many case statements.
- **`break`** exists the **`switch`** block
- **`STATEMENT`** usually returns **`int`** or **`enum`** value

While loop

```
while (STATEMENT) {
          // Loop while STATEMENT == true - typically condition is changed within the loop
}
```

- Usually used when the exact number of iterations unknown a priori
- Easy to form an endless loop by mistake

For loop

```
for (INITIAL CONDITION; END_CONDITION; INCREMENT) {
          // This happens until END_CONDITION == false
}
```

- In C++ **`for`** loops are – comparably – fast.
- Less flexible than **`while`** but less error-prone
- Use **`for`** when number of iterations is fixed and **`while`** otherwise

Range for Loop

- Iterating over standard containers like **`array`** or **`vector`** has simpler syntax
- Avoid mistakes with indices
- Show intent with syntax
- Has been added in C++

```
for (const auto& value : container) {
          // This happens for each value in the container
}
```

Exit loops and iterations

- We have control over loop iterations
- Use **`break`** to exit the loop
- Use **`continue`** to skip to next iteration

```
while (true) {
     int i = /* Magically get new int */
     if ( i % 2 == 0) {
          cerr << I << endl;
     } else {
          break;
     }
}
```

Thank you