ADVANCED PROGRAMMING IN C++

Basics

Patrick Bader · SS 2023

Hello World

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
  cout << "Hello World!\r\n";
  return 0;
}
```

Input / Output

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

int main(int argc, char* argv[])
{
  std::cout << "Type in your name:\r\n";
  std::string name;
  std::cin >> name;
  std::cout << "Hello " + name;

  return 0;
}
```

Namespaces

Prevent name collisions

Standard library uses std namespace

Defining names in a namespace:

```
namespace my_space {
// add declarations here
}
```

Using namespaces:

- Direct:

  ```
  std::cout;
  ```

- Import single names:

  ```
  using std::cout; // cout usable without std::
  ```

- Import entire namespace:

  ```
  using namespace std; // no std:: necessary
  ```

Strings

Use string class in the standard library: `#include <string>`

Variable definition: `std::string first_name = "Jon";`

Length: `int len = first_name.length();`

Concatenation: `std::string name = first_name + " Doe";`

Character access:

- Read single character: `char initial = name[0];`
- Change single character: `name[0] = 'R';`

Output: `std::cout << name;`

Important: No bounds checking is performed with the index operator `[]`. If you want bounds checking, use the .at() method: `name.at(0) = 'R';`

Dynamic Arrays

Vector class in the standard library: `#include <vector>`

Variable definition: `std::vector<int> v = { 1, 2, 3, 4 };`

Get current size: `int s = v.size();`

Appending: `v.push_back(4);`

Removing last Element: `v.pop_back();`

Single element access:

- Read single element: `int i = v[3];`
- Change single element: `v[1] = 100;`

Important: No bounds checking is performed with the index operator `[]`. If you want bounds checking, use the .at() method: `v.at(0) = 100;`

I/O Streams

I/O stream classes in the standard library:

```
#include <iostream>
```

Write to standard output:

```
int x = 10; float y = 3.1f;
std::cout << x;
std::cout << y;
std::cout << endl;
// equivalent with chaining:
std::cout << x << y << endl;
```

Read from standard input:

```
std::cin >> y;
```

Control Structures - Branches

```
if (5 < 10)
{
  std::cout << "five is smaller";
}
else
{
  std::cout << "five is larger or equal";
}

if (int i = 9; 5 < 10)
{
  std::cout << "five is smaller";
}
```

Control Structures - Branches

```
int i = 10;
int j;

switch (i) {
  case 2:
  case 4:
    j = 9;
    break;
  case 6:
    j = 2;
    break;
  default:
    j = 11;
}
```

Control Structures - Loops

```
int i = 10;
while (i >= 0) {
  std::cout << "Count down: " << i;
  --i;
}

do {
  std::cout << "Condition at bottom";
} while (false);

for (int i = 0; i < 10; ++i) {
  std::cout << "Iteration: " << i;
}

std::vector<int> v{ 1, 3, 7, 8 };
for (int e : v) {
  std::cout << e;
}
```

(Free) Functions

Free functions do not belong to class instances

Are usually declared in the global scope or in a namespace

Behave like static methods in Java

Declaration:

```
int add(int a, int b);
```

Definition:

```
int add(int a, int b)
{
  return a + b;
}
```

Functions at least have to be declared before they are used.

Primitive Data Types - Integer

Integer types:

```
char c; short s; int i; long l; long long ll;
```

Sizes are compiler dependent: `char` ≤ `short` ≤ `int` ≤ `long` ≤ `long long`

Initialized and used as in Java:

```
int x = 10;
```

There are unsigned versions:

```
unsigned int ui = 3;
```

Don't mix these with signed for arithmetic → unexpected results.

Primitive Data Types - Bool

```
  bool b = true;
```

`false` implicitly casts to integer value `0`

`true` implicitly casts to integer value `1`

Integer value `0` implicitly casts to `false`

All other integer values cast to `true`

Primitive Data Types - Floating Point

Floating point types:

```
float f; double d; long double ld;
```

Formats are fixed for `float` (IEEE-754 32 bit) and `double` (IEEE-754 64 bit). `long double` is implementation defined, usually 80 bit.

Initialized and used as in Java:

```
float y = 3.1f;
```

Conversions between integer and floating point types are done implicitly in C++!

Primitive Data Types - Enumerations

```
enum class Answer { Yes, No };
```

will probably be covered later in the course

Primitive Data Types - Indirection

Pointers:

```
float* ptr;
```

Contain the memory address at which some value of the respective type is located

References:

```
float f = 5.0f;
float& g = f;
```

Similar to pointers, however: always refer to a valid memory location Have to be assigned when declared and cannot be reassigned