CH-230-A

# Programming in C and C++

C/C++

## Tutorial 9

Dr. Kinga Lipskoch

Fall 2022

# `cin` : Console Input (1)

- `cin` is the companion stream of `cout` and provides a way to get input
  - as `cout`, it is declared in `iostream`
- The overloaded operator >> (extractor) gets data from the stream

```
float f;
cin >> f;
```

  - Warning: it does not remove endlines
  - If you are reading both numbers and strings you have to pay attention

# Boolean and String as Types

- `bool` as distinct type
  (also now in C, you need to include `stdbool.h`)

```
bool c;
c = true;
cout << c << endl;
```

- `string` as distinct type

```
string s;
s = "Hello, I am a C++ string";
cout << s << endl;
```

# cin : Console Input (2)

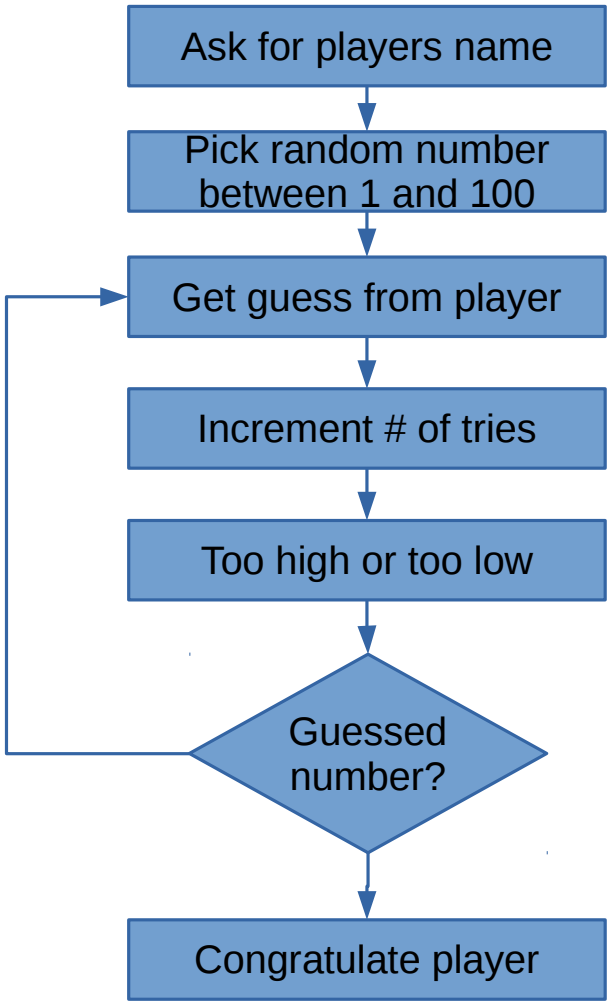
- There is one `getline` function and one `getline` method
- The function `getline` is a global function and reads a string from an input stream

```
string str;
getline(cin, str);
```

- The method `getline` gets a whole line of text (ended by '\n' and it removes the separator)
- It reads a C string (a character array that ends with a '\0')

```
char buf[50];
string s;
cin.getline(buf, 50);
s = string(buf);
// convert to a C++ string
```

# A Simple Guessing Game



# How to Pick a Random Number

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4  using namespace std;
5  int main() {
6    int die;
7    int count = 0;
8    int randomNumber;
9    // init random number generator
10   srand(static_cast<unsigned int>(time(0)));
11   while (count < 10) {
12     count++;
13     randomNumber = rand();
14     die = (randomNumber % 6) + 1;
15     cout << count << ": " << die << endl;
16   }
17   return 0;
18 }
```

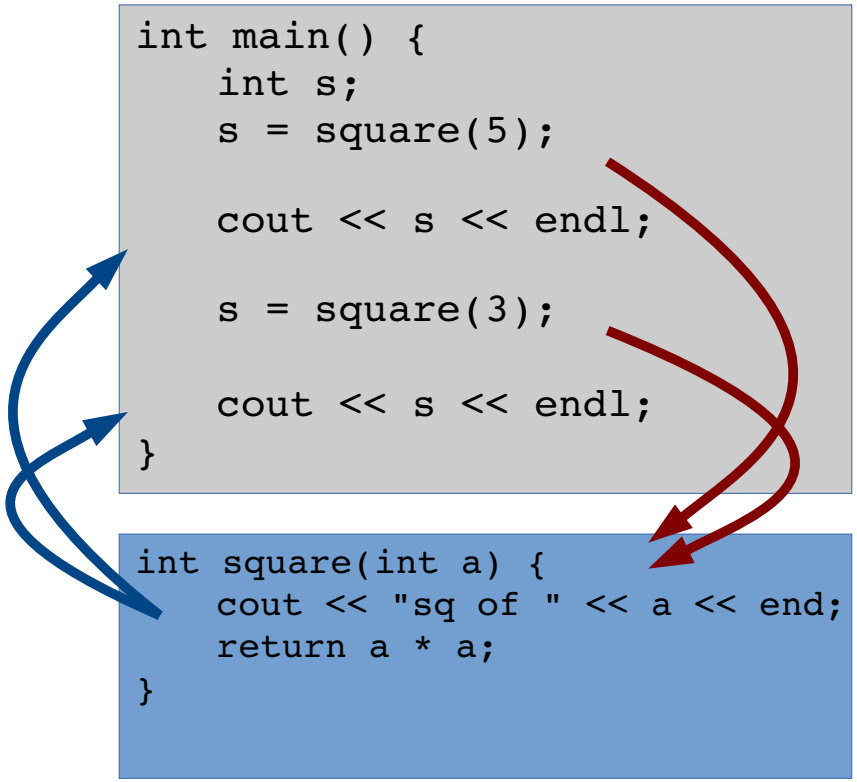
# C++ Extensions to C

- Inline functions
  - available in C since the standard C99
- Overloading
- Variables can be declared anywhere
  - possible in C since the standard C99
- References

# Inline Functions (1)

- For each call to a function you need to setup registers (setup stack), jump to new code, execute code in function and jump back
- To save execution time macros (i.e., `#define`) have often been used in C
- A preprocessor does basically string replacement
- Disadvantage: it is error prone, no type information
- `inline.cpp`

# Inline Functions (2)




```
int main() {
    int s;
    cout << "sq of " << 5 << end;
    s = 5 * 5;

    cout << s << endl;

    cout << "sq of " << 3 << end;
    s = 3 * 3;

    cout << s << endl;
}
```

# Function Overloading

```
1  #include <iostream>
2  using namespace std;
3  int division(int dividend, int divisor) {
4    return dividend / divisor;
5  }
6  float division(float dividend, float divisor) {
7    return dividend / divisor;
8  }
9  int main() {
10   int ia = 10;
11   int ib = 3;
12   float fa = 10.0;
13   float fb = 3.0;
14
15   cout << division(ia, ib) << endl;
16   cout << division(fa, fb) << endl;
17   return 0;
18 }
```

Output: 3 3.33333

# Variable Declaration "Everywhere"

```
1 void function() {
2    .......
3    printf("C-statements...\n");
4    ......
5    int x = 5;
6    // now allowed, works in C
7    // as well since standard C99
8 }
```

# No "Real" References in C (1)

Accessing a variable in C

- `int a;` `// variable of type integer`
- `int b = 9;` `// initialized variable of type integer`
- `a = b;` `// assign one variable to another`
- `b = 5;` `// assignment of value to variable`

# No "Real" References in C (2)

Accessing variable via pointers

- `int a;` `// variable of type integer`
- `int b = 5;` `// initialized variable`
- `int* ptr;` `// pointer to integer`
- `ptr = &a;` `// address of a is assigned to pt`
  `// (it points to a)`
- `*ptr = b;` `// assign b to content where ptr`
  `// points to a is now 5`

# References in C++

A reference can be seen as additional name or as an alias of the variable

- `int a;`
- `int b = 5;` `// initialized variable`
- `int& ref = a;` `// reference to variable of type int`
- `ref = 7;` `// assignment of variable a via reference ref`

# "Real" Call-by-Reference (1)

```
1  #include <stdio.h>
2  void swap_cpp(int &a, int &b);    // prototype
3  void swap_c(int *a, int *b);      // prototype
4  void swap_wrong(int a, int b);    // prototype
5  int main(void) {
6    int a_cpp = 3, b_cpp = 5,
7    a_c = 3, b_c = 5,
8    a = 3, b = 5;
9    swap_cpp(a_cpp, b_cpp);
10   swap_c(&a_c, &b_c);
11   swap_wrong(a, b);
12   printf("C++: a=%d, b=%d\n", a_cpp, b_cpp);
13   printf("C: a=%d, b=%d\n", a_c, b_c);
14   printf("Wrong: a=%d, b=%d\n", a, b);
15   return 0;
16 }
```

# "Real" Call-by-Reference (2)

```
1  void swap_cpp(int &a, int &b) {
2    // real Call-by-Reference
3    int help = a;
4    a = b;
5    b = help;
6  }
7  void swap_c(int *a, int *b) {
8    // not real Call-by-Reference
9    // Call-by-Value via Pointer

10   int help = *a;
11   *a = *b;
12   *b = help;
13 }
14 void swap_wrong(int a, int b) {
15   // Call-by-Value
16   int help = a;      // no swapping of passed
17   a = b;             // parameters,
18   b = help;          // since only copies are swapped
19 }
```

# Constant References

- References are not only useful if arguments are to be modified
- No copying of (possibly large) data objects will happen
- Using references saves time
- To show that parameters are not going to be modified constant references should be used

```
void writeout(const int &a, const int &b) { ... }
```

- `ref_timing.cpp`

# Dynamic Memory Allocation

C++ has an operator for dynamic memory allocation

- It replaces the use of the C `malloc` functions
- `alloc_in_c.c`
  - Easier and safer
- The operator is called `new`
  - It can be applied both to user defined types (classes) and to native types
    - `operator_new.cpp`
    - use `-std=c++0x` switch to compile program according to the standard C++11
    - use `-std=c++14` switch to compile program according to the standard C++14

# Operators `new` and `delete`

- `new`
  - primitive types are initialized to 0
  - returned type is a pointer to the allocated type
- `delete` releases allocated memory
  - `delete ptr_1;  // releases int`
  - `delete [] ptr_7;  // releases int-array`
- Memory that has been allocated via `new []` must be released by `delete []`
- C: `malloc() --> free()`
- C++: `new --> delete`