Advanced Programming in C++

Kristof Van Laerhoven
kvl@eti.uni-siegen.de

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

using namespace std;

int main() {
  vector<string> msg {"Welcome", "to", "advanced", "C++!"};
  for (const string& word : msg) {  // C++11 standard
    cout << word << " ";
  }
  cout << endl;
}
```

Contents

This course in a nutshell

In this course, you learn advanced themes in C++ programming

The course consists of:

- Lecture: 2h per week, basic concepts
- Lab: 2h per week, getting / correcting programming tasks
- Homework: ± 2h per week, solving assignments

This course in a nutshell

Links to lecture slides and assignments are available on moodle:

- https://moodle.uni-siegen.de/course/view.php?id=34345
- updates to the slides will be made available during the term

We learn C++ by doing, hence: *follow lectures and exercises*

Disclaimer:
This course's material was inspired by similar course from Prof. Roland Wismueller (Uni Siegen) and Prof. Hannah Bast (Uni Freiburg)

Examination and grading

50% of your grade comes from:

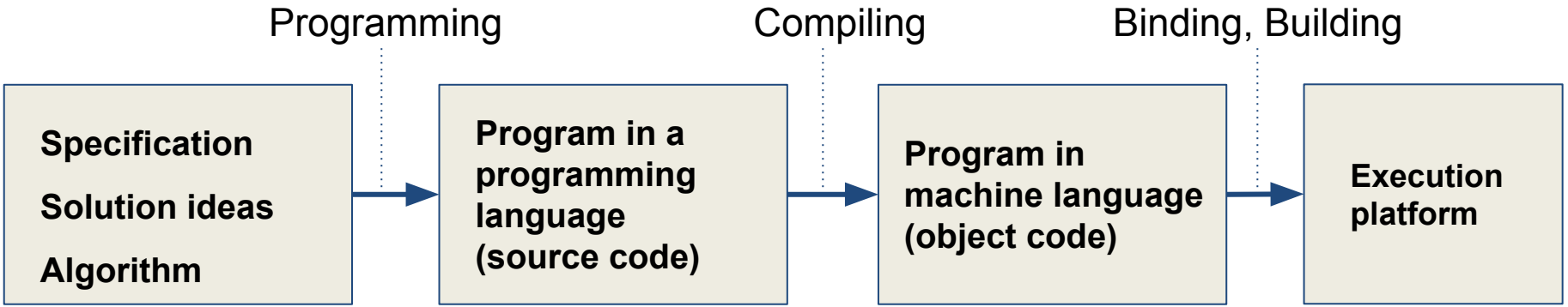
- Set of programming assignments during term and programming on paper
- These need to be delivered in-class
  => your presence is required in these sessions

50% of your grade comes from the 1-hour written exam:

- 1 handwritten A4 (double-sided) page is allowed
- Bring a photo ID, your student ID and a pen (blue/black ink only)
- Structure: Programming tasks *very similar* to the exercises

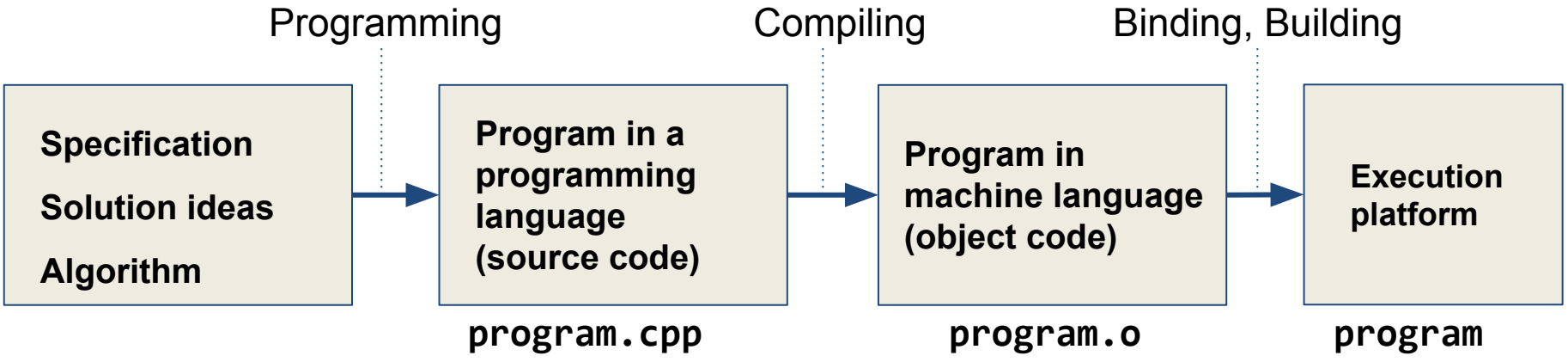
1. Designing programs

Steps in creating a program:



1. Designing programs

Steps in creating a program:



Editor:
`nano program.cpp`

GCC compiler:
`g++ program.cpp`

`./program`

1. Designing programs

1.1. Program structure:

- A program is essentially a series of *machine instructions* that tell the system what to do, step by step, similar to a recipe
- Source code allows humans to formulate instructions in an understandable fashion, which then can be translated to machine instructions → In this course, we use C++ to create source code
  - C++ source code files are text files that end with **.cpp** or **.h**
  - A *compiler* then translates these to a program that consists out of machine instructions
- Creating source code needs a file system and operating system

1. Designing programs

1.2. Using GitHub Repository, g++, editors

We will use a github repository: https://github.com/kristofvl/AdvancedCPP as a code base for the slides, all exercises, and home works from the lecture

- You will need a github account
- It allows keep track of changes, and distributing larger projects

We'll use GNU g++. For Linux or macOS, you can directly use g++
In Windows, you can install Windows Subsystem for Linux (WSL), to have a Linux environment to use g++ inside it

1. Designing programs

1.2. Using GitHub Repository, g++, editors

You are free in the use of editor. Here are a few options:

Microsoft Visual Studio Code (VSCode) , Sublime , Lapce, Zed

Text-based coding editors: Vim, Emacs, NeoVim, Helix

Not suggested: Notepad, Gedit, and other similar editors

1. Designing programs

BDay.cpp

```
/* The Birthday Paradox -- an illustration
   Author: kvl,   Date: first week     */



int main() {



  return 0;
}
```

1. Every program should mention who programmed it

2. Every program needs a function "main" that contains all code to be executed

3. The function "main" will exit and send a number (usually zero, for "no error") to the operating system

1. Designing programs

BDay.cpp

```
/* The Birthday Paradox -- an illustration
   Author: kvl,   Date: first week     */
#include <iostream>
class BDay {
 /* p(x) --> probability of x happening,  p(x) = 1 - p(not x)
    p(2 persons have same birthday) =  1 - (365-1)/365 * (365-2)/365 * ... * (365 - (n-1))/365 */
 public:
  double prob(int n);
};
double BDay::prob(int n) {
  double p = 1.0;   // probability that out of n people, 2 have the same birthday
  for (int i = 0; i < n; i++) {  // i = 1, 2, ..., n-1
     p = p * (365-i)/365;
  }
  return 1 - p;
}
int main() {
  int n = 23;
  BDayP b;  // create an object to access the prob() method:
  std::cout << "The chance that 2 out of " << n << " people have a same birthday is " << b.prob(n);
  return 0;
}
```

1.3. Compiling and building a program:

- Type `g++` to launch the GCC c++ compiler: `g++ BDay.cpp`
  - `g++` creates the program, a file with the default name `a.out` in the current directory, which can be executed in the terminal: `./a.out`
- Add `-o` to specify another name: `g++ BDay.cpp -o BDay`
  - `g++` creates the file `BDay` , which can be executed: `./BDay`
- The compiler tries to compile all other needed source files and needs a `main` function in the files that specifies what your program does
  - `g++` can compile multiple files, e.g.: `g++ ex1.cpp ex2.cpp`
  - `g++` can compile code for later use in a program with `-c` :
    `g++ -c BDay.cpp`, which creates `BDay.o`
    `g++ BDayx.o` then creates the program `a.out`

1. Designing programs

1.3. Compiling and building a program:

- Add `--std` to specify which C++ standard your code is for. If for instance you use features for C++11 you need to add `--std=c++11`:

```
g++ BDay.cpp -o BDay --std=c++11
```

  Features you use may be in different standards, and you can specify them by the option `--std` to let the compiler compile the source code in different standards

1.4. Including libraries:

- The compiler can include others' code as a library that is mentioned in the source code (for example `#include <iostream>`)
  - In this course, we will see mostly such *standard* libraries, `g++` knows where to search for their files and links these:

```
g++ BDaySimple.cpp
```

BDaySimple.cpp

```cpp
/* The Birthday Paradox -- a simplification */
#include <iostream>
int main() {
  std::cout << "The chance that 2 out of 23 people have";
  std::cout << " a same birthday is about 0.5" << std::endl;
  return 0;
}
```

1.4. Including libraries:

- The compiler can include others' code as a library that is mentioned in the source code (for example ncurses: `#include <ncurses.h>`)
  - Other non-standard libraries need to be linked explicitly with `-l`:

```
g++ BDayCentre.cpp -l ncurses
```

BDayCentre.cpp

```
/* The Birthday Paradox -- in the *middle*  */
#include <ncurses.h>  // draw text in terminal screen
int main() {
  initscr();    // initialize ncurses window
  mvaddstr(LINES/2,   COLS/3, "The chance that 2 out of 23 have");
  mvaddstr(LINES/2+1, COLS/3, "the same birthday is about 0.5. wow.");
  getch();      // capture the user's pressed key
  endwin();     // close the ncurses window
  return 0;
}
```

1. Designing programs

1.5. Indenting your code:

- To make everyone's code look the same, we recommend using **cpplint**
- We *also* recommend **2-space indentation** (see *all* examples in slideset):

```
void myFunction(bool exec, bool size) {  // indent after each {
◇◇int ret = 0;
◇◇if (exec) {  // indent after each {
◇◇◇◇for (int i = 0; i < size; i++) {  // indent after each {
◇◇◇◇◇◇ret += i;
◇◇◇◇}  // de-indent before each }
◇◇} else {  // indent after each {
◇◇◇◇ret = 12;
◇◇}  // de-indent before each }
}  // de-indent before each }
```

2. Data: Variables, Types, Constants

2.1. The basic components of a program:

- Reserved keywords
- Preprocessor directives
- Names
- Constants
- Operators
- Braces
- Separators
- Comments

```
/* The Birthday Paradox -- in short */
#include <iostream>
int main() {
  std::cout << "The chance that 2 out of";
  std::cout << " 23 people have a same";
  std::cout << " birthday is about ";
  std::cout << 0.5 << std::endl;
  return 0;  // 0 back to operating system
}
```

2.1.1. Reserved C++ keywords

Reserved keywords are words that are reserved for special meaning by the language standard and cannot be used as identifiers (names for variables, functions, classes, etc.). E.g.:

`bool`	`do`	`namespace`	`switch`
`break`	`double`	`new`	`this`
`case`	`else`	`private`	`true`
`catch`	`false`	`protected`	`using`
`char`	`float`	`public`	`virtual`
`class`	`for`	`return`	`void`
`const`	`if`	`short`	`while`
`delete`	`int`	`sizeof`	

2.1.2. Preprocessor directives

Source code can also contain so-called *preprocessor directives* that start with a `#` : We will use solely:

- `#include`, followed by a source file either:
  - surrounded by `<` and `>` , for example: `#include <ncurses.h>` for source code from standard libraries
  - surrounded by `"` and `"` , for example: `#include "myCode"` for source code in the current directory
- **header guards** to ensure that code is included only once:

```
#ifndef HEADERFILE
#define HEADERFILE

#endif
```

2.1.3. Names:

You can define names to variables, parameters, functions, etc.

- these names need to be unique (so no keywords)
- they can contain only letters, digits and underscores (_)
- they need to start with letters or an underscore
- their length is nearly unlimited
- examples:

correct	wrong	reason
`Sum`	`get Name`	no empty spaces
`getName`	`Ver2-1`	minus '-' not allowed
`_all4you`	`2exp4`	starts with a digit
`__1_2`	`while`	C++ keyword

2. Data: Variables, Types, Constants

2.1.4. Constants:

`15` (integer) `3.14159f` (float) `3.14159` (double)
`'p'` (character) `"brb"` (string) `true` (boolean)

2.1.5. Operators:

`+ - * / && || = == >= <= << >> < > ...`

2.1.6. Braces:

`( ) [ ] { }`

2.1.7. Separators:

`, ; .` (space) as well as tabs or new lines

2. Data: Variables, Types, Constants

2.1.8. Comments:

- `// comment till the end of the line`
- ```
  /* comment that spans across
     multiple lines */
  ```
- note that a multi-line comment cannot contain */:
  ```
  /* this example comment */ would
     cause an error */
  ```

```
// probability that out of n people, 2

/* p(x) --> probability of x happening
   p(x) = 1 - p(not x)
   p(2 persons have same birthday) =
...*/

/** The Birthday Paradox -- illustration
*   Author:  kvl
*   Date:    last week
*/
```

IMPORTANT: Comments should explain your code but never be trivial

good: `int scrMaxWidth; // maximum screen width`

bad: `float aspectRatio; // this variable holds the aspect ratio`
```

2. Data: Variables, Types, Constants

2.1. Example:

```
/* An interactive example */
#include <iostream>
int main() {
  char name[80];  // symbols array for the user's name
  std::cout << "Hi, what's your name?" << std::endl;
  std::cin >> name;  // read the name from terminal
  std::cout << "Welcome " << name;
  if (name[0] == 'K') {
    std::cout << ", I like your name!"
  }
  std::cout << std::endl;
  return 0;  // return a zero
}
```

comments

preprocessor

separators

operators

braces

constants

names

keywords

2.2. Variables:

- represent *memory space*, where data of a certain type can be stored
- need to be declared and ideally initialized before use:
```
int keyPressCounter = 0;  // how often did user press a key?
```
- have values that after declaration can be 'read out' and changed:
```
if (keyPressCounter > 27)  // after 27 key presses,
  keyPressCounter = 0;     // we set it back to zero
```
- can't be changed afterwards, when declared (and initialized) as **constants**: `const int answer = 42; // after this, answer stays 42`
- live in a certain *scope*, typically the function in which it was declared. After this function ends, the variable is deleted from memory.

2.3. Data types: Integral

Native Type	Bytes	Range	Fixed width types <cstdint>
bool	1	true, false	
char †	1	implementation defined	
signed char	1	-128 to 127	int8_t
unsigned char	1	0 to 255	uint8_t
short	2	-2^{15} to $2^{15}-1$	int16_t
unsigned short	2	0 to $2^{16}-1$	uint16_t
int	4	-2^{31} to $2^{31}-1$	int32_t
unsigned int	4	0 to $2^{32}-1$	uint32_t
long int	4/8*		int32_t/int64_t
long unsigned int	4/8*		uint32_t/uint64_t
long long int	8	-2^{63} to $2^{63}-1$	int64_t
long long unsigned int	8	0 to $2^{64}-1$	uint64_t

2.3. Data types: Floating-Point

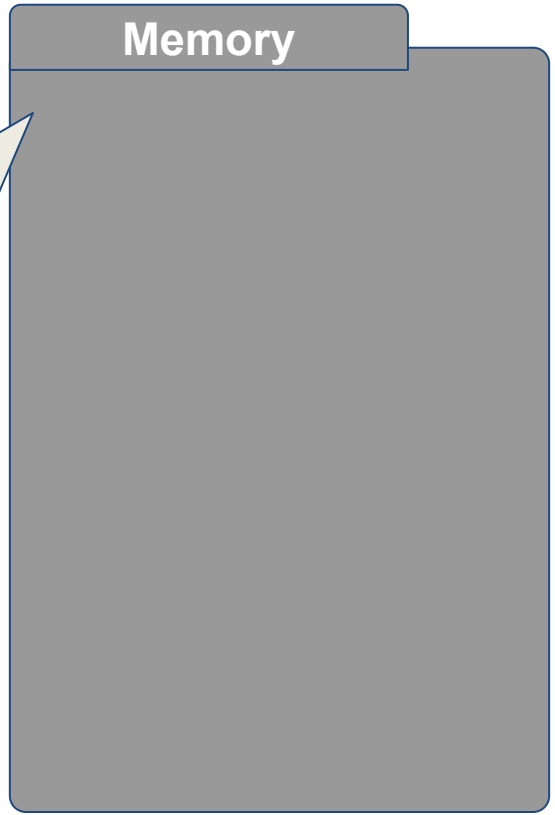
Native Type	IEEE	Bytes	Range	Fixed width types C++23 <stdfloat>
(bfloat16)	N	2	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$	std::bfloat16_t
(float16)	Y	2	0.00006 to 65,536	std::float16_t
`float`	Y	4	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$	std::float32_t
`double`	Y	8	$\pm 2.23 \times 10^{-308}$ to $\pm 1.8 \times 10^{+308}$	std::float64_t

2.3. Data types

- A type defines: what *values* the variable can have,
  how much memory is allocated for it, and
  which *operations* are possible
- All variables and constants have a type in C++
  - for constants, you can tell the type by their form
  - for variables, this is explicit in the declaration

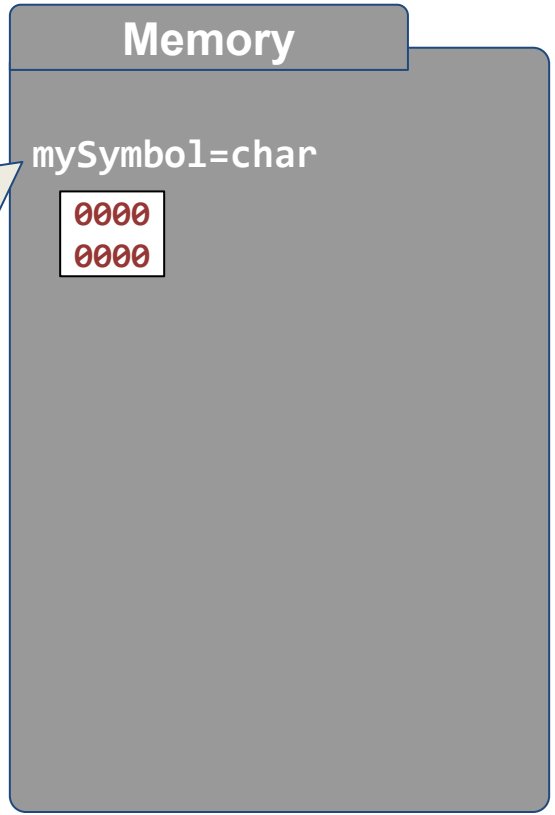
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;   // store one character
  int myInteger;   // store an integer
  bool myBoolean;  // store a boolean
  float myFloat;    // store a floating point
  myInteger = 12;  // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';  // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```



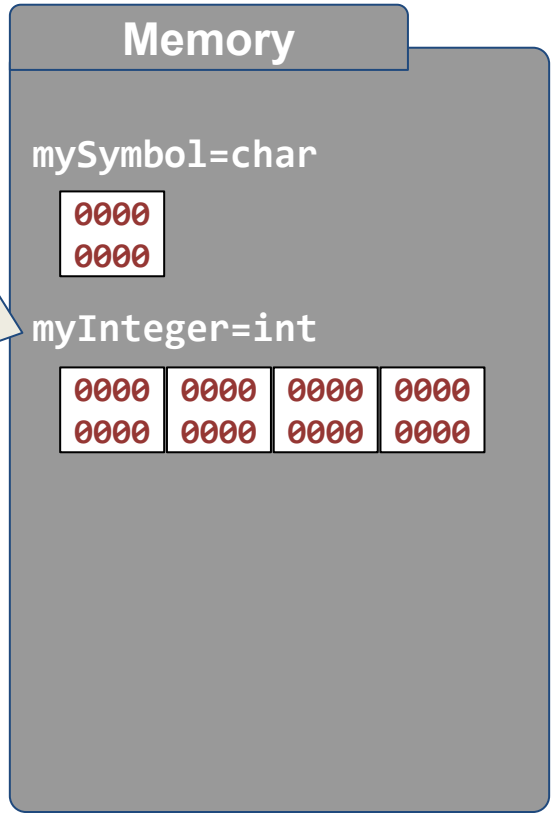
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;   // store one character
  int myInteger;   // store an integer
  bool myBoolean;  // store a boolean
  float myFloat;    // store a floating point
  myInteger = 12;  // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';  // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```



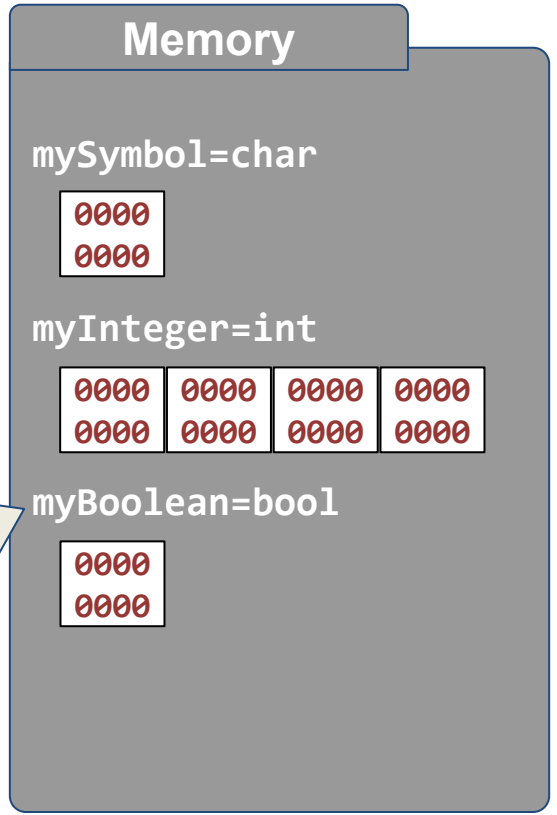
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;   // store one character
  int myInteger;   // store an integer
  bool myBoolean;  // store a boolean
  float myFloat;    // store a floating point
  myInteger = 12;  // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';  // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```



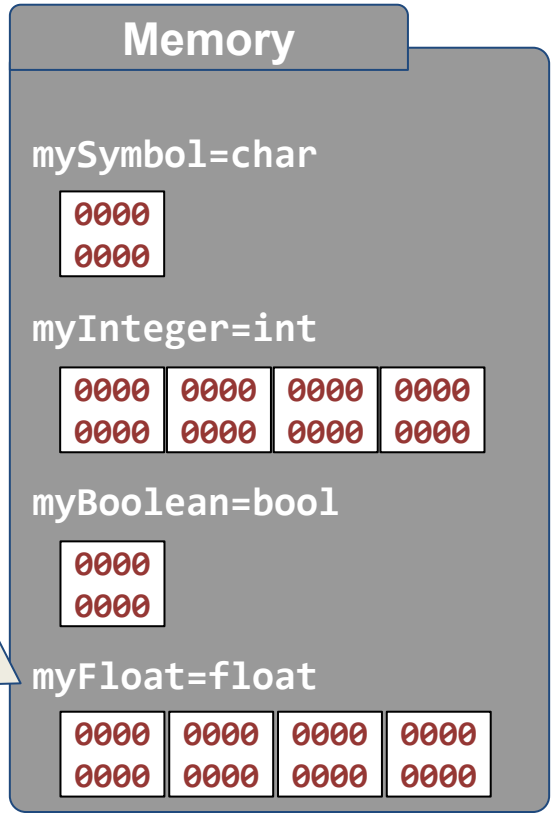
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;   // store one character
  int myInteger;   // store an integer
  bool myBoolean;  // store a boolean
  float myFloat;    // store a floating point
  myInteger = 12;  // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';  // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```



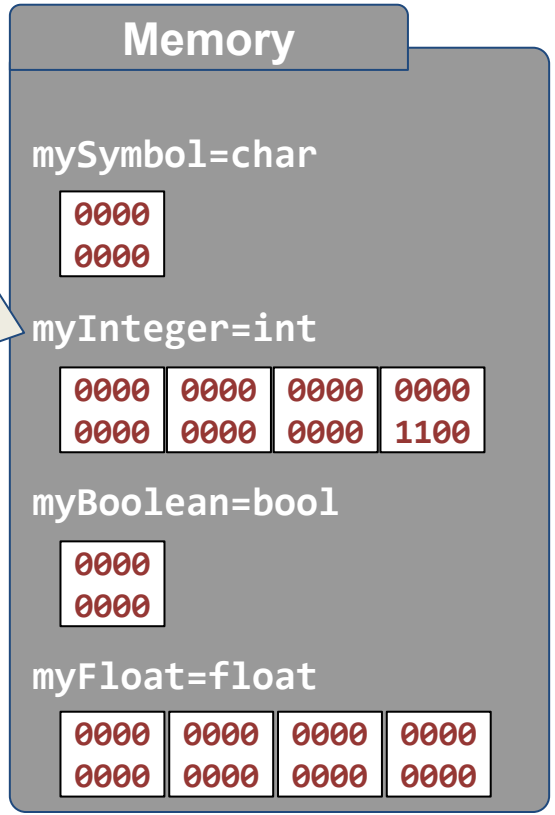
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;   // store one character
  int myInteger;   // store an integer
  bool myBoolean;  // store a boolean
  float myFloat;    // store a floating point
  myInteger = 12;  // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';  // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```



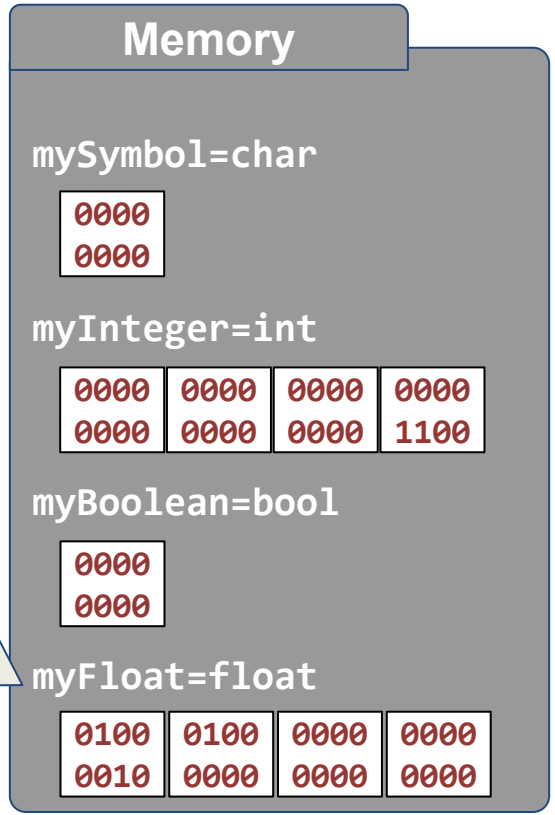
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;   // store one character
  int myInteger;   // store an integer
  bool myBoolean;  // store a boolean
  float myFloat;    // store a floating point
  myInteger = 12;  // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';  // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```



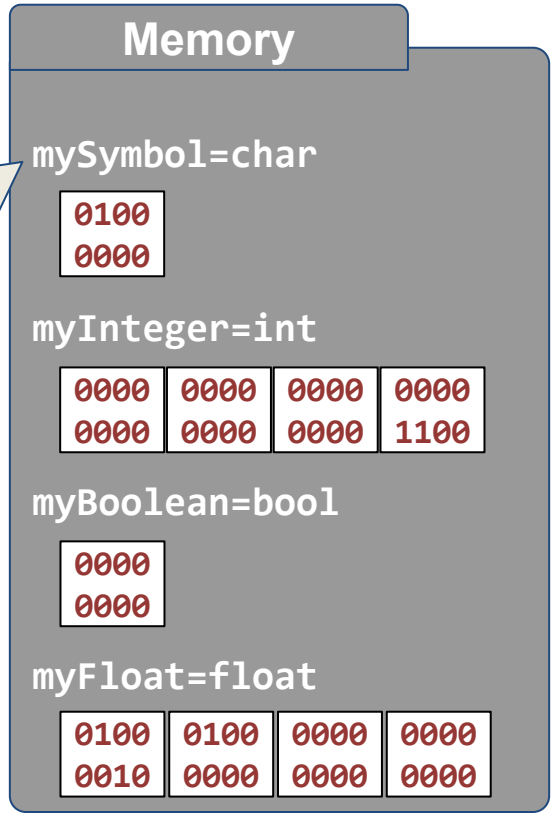
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;   // store one character
  int myInteger;   // store an integer
  bool myBoolean;  // store a boolean
  float myFloat;    // store a floating point
  myInteger = 12;  // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';  // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```



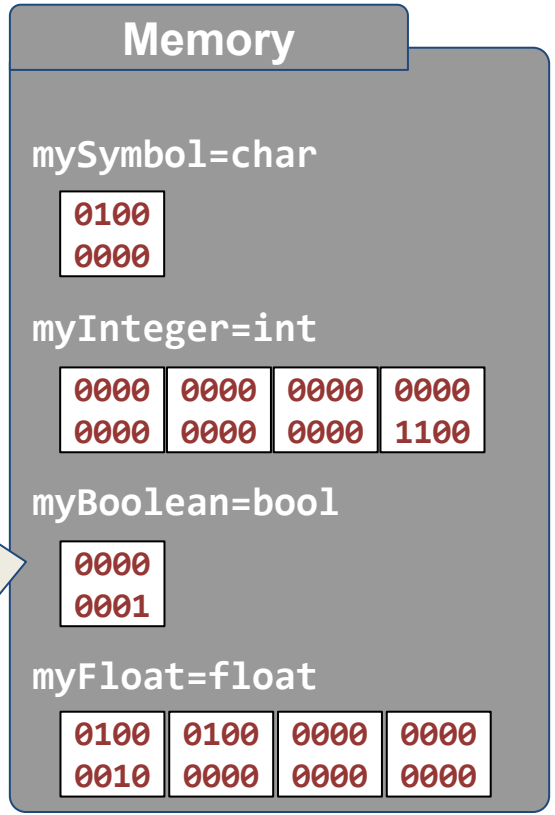
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;   // store one character
  int myInteger;   // store an integer
  bool myBoolean;  // store a boolean
  float myFloat;    // store a floating point
  myInteger = 12;  // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';  // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```



2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;   // store one character
  int myInteger;   // store an integer
  bool myBoolean;  // store a boolean
  float myFloat;    // store a floating point
  myInteger = 12;  // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';  // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```



2.3. Data types and variables

- There are few basic types in C++:
  - `int` stored in 4 bytes, range from -2147483648 to 2147483647
  - `float` stored in 4 bytes (IEEE754), range from -3.4 x 10^{38} to 3.4 x 10^{38}
  - `double` stored in 8 bytes (IEEE754), range from -1.7 x 10^{308} to 1.7 x 10^{308}
  - `char` stored in 1 byte (see ASCII table)
  - `bool` stored in 1 byte
- **Constants** for these type are visible from how they are written:
  - `int` whole numbers, e.g.: `729`, `-3628`, `0`, `-12632832`
  - `float` numbers with decimal range and `f`, e.g.: `3.612f`, `5.2f`, `0.001f`
  - `double` numbers with decimal range, e.g.: `3.612`, `5.2`, `0.001`
  - `char` a character between single quotes, e.g.: `'?'`
  - `bool` either `true` or `false`

2.3. Data types and variables

Type	SUFFIX	Example	Notes
`int`	/	2	
`unsigned int`	u, U	3u	
`long int`	l, L	8L	
`long unsigned`	ul, UL	2ul	
`long long int`	ll, LL	4ll	
`long long unsigned int`	ull, ULL	7ULL	
`float`	f, F	3.0f	only decimal numbers
`double`		3.0	only decimal numbers

C++23 Type	SUFFIX	Example	Notes
`std::bfloat16_t`	bf16, BF16	3.0bf16	only decimal numbers
`std::float16_t`	f16, F16	3.0f16	only decimal numbers
`std::float32_t`	f32, F32	3.0f32	only decimal numbers
`std::float64_t`	f64, F64	3.0f64	only decimal numbers
`std::float128_t`	f128, F128	3.0f128	only decimal numbers

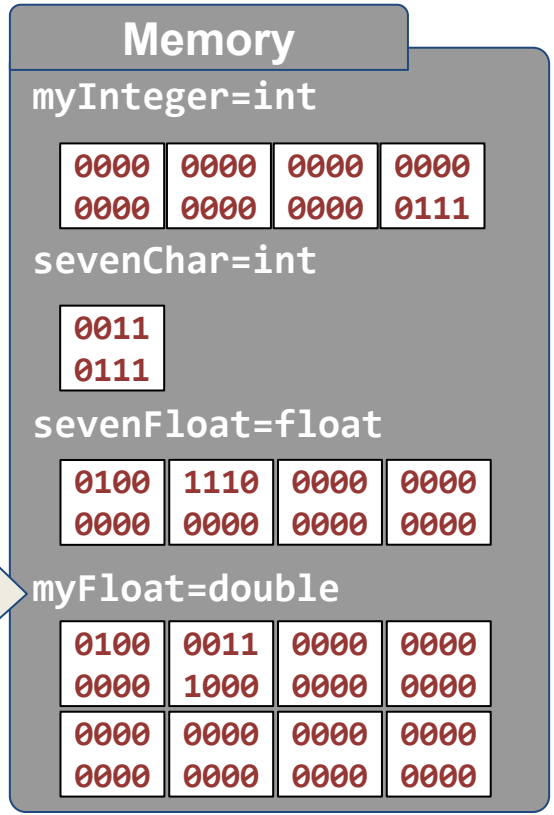
2.3. Data types and variables

Representation	PREFIX	Example
`Binary` C++14	`0b`	`0b010101`
`Octal`	`0`	`0307`
`Hexadecimal`	`0x` or `0X`	`0xFFA010`

C++14 also allows *digit separators* for improving the readability `1'000'000`

2.3. Data types: Impact in memory

```
/* reserving variables */
int main() {
  int sevenInteger = 7;      // seven as integer
  char sevenChar = '7';      // seven as character
  float sevenFloat = 7.0f;   // seven as float
  double sevenDouble = 7.0; // seven as double
  return 0;
}
```



2.3. Data types and variables

When you need a variable, you need to **declare** it first (and you can give them a value straight away, this is called **initialization**):

```
char mySymbol = '&';     // store one character and set it to '&'
int myNumber;            // store an integer, no initial value
bool areWeDone = false; // stores a boolean, set to false
float heightInMeters = 1.85f; // stores 1.85 as a float
double highPrecision;    // stores a double floating point number
```

Later, you can give variables **new values**:

```
areWeDone = true;   // we have now set this variable to true
highPrecision = 0.0;  // we have now set this variable to 0.0
```

Multiple variables can be declared at once: `int myNum = 3, yourNum = 8;`

2.3. Data types and variables

Example: Initialize a character to `a`, show its value in the terminal, then change the value to a `q` , then show it in the terminal again.

```
#include <iostream>  // to allow use of std::cout and std::endl
int main() {
  char mySymbol;   // variable to hold a character
  mySymbol = 'a';
  std::cout << mySymbol << std::endl;  // prints out "a" and endline
  mySymbol = 'q';
  std::cout << mySymbol << std::endl;  // prints out "q" and endline
  return 0;
}
```

2.3. Data types and variables: auto

C++11 The `auto` keyword specifies that the type of the variable will be automatically deduced by the compiler (from its initializer)

```
auto a = 1 + 2;   // 1 is int, 2 is int, 1 + 2 is int!
//    -> 'a' is "int"
auto b = 1 + 2.0; // 1 is int, 2.0 is double. 1 + 2.0 is double
//    -> 'b' is "double"
```

`auto` can be very useful for maintainability and for hiding complex type definitions

```
for (auto i = k; i < size; i++)
    ...
```

On the other hand, it may make the code less readable if excessively used because of type hiding

Example: `auto x = 0;` in general makes no sense (`x` is `int`)

2.4. Variables and scope

- A variable only exists within a **scope** (block of code, usually between curly braces `{` and `}`) and is deleted after the scope ends (after a `}`)
- Variable names cannot use names that are already taken

```
{  // from here starts one scope:
    int a = 3, b = 1;
    {          // from here starts another scope:
        c = a + b;     // error: c doesn't exist here yet
        int c = 0;
        c += a;        // this works, c is defined here
    }
    b = c;           // error: c doesn't exist here
    double a = 2.1; // error: a already taken
}
```

2.5. Type conversions

- Variables can sometimes take values from other variables and constants that do not have the same type, **implicitly** converting them
- We'll stick to **explicit conversion**, using the type in braces:

```
double myDouble = 1.1;
int aNumber = 5;
char c = 'a';
bool b = true;
myDouble = (double)aNumber;  // this works: myDouble == 5.0
myDouble = (double)c;        // this works: myDouble == 97.0
aNumber = (int)myDouble;     // works:      aNumber == 97
c = (char)98.0f;             // works too:  c == 'b'
```

2.5. Type conversions

Example: Find out what the ASCII-codes are for the symbols '?', '&', and '#'

```
#include <iostream>  // to allow use of std::cout and std::endl
int main() {
  char question = '?';   // three variables to hold characters
  char ampersand = '&';   // variable to hold a character
  char hash = '#';   // variable to hold a character
  std::cout << (int)question << std::endl;   // print ASCII code for '?'
  std::cout << (int)ampersand << std::endl;  // print ASCII code for '&'
  std::cout << (int)hash << std::endl;       // print ASCII code for '#'
  return 0;
}
```

🏠 Homework: Can you write this program in a more memory-efficient way?

3. Basic statements: if, switch, loops

3.1. Statements and assignments:

- Variables are the program's data, statements specify what to do with the data
- Statements *always* end with `;` and are executed sequentially
- Assignments *always* calculate the value of what is right of `=` and return this:

```
int x, y, mult;
bool b;
x = 2;                  // assignment returns 2
mult = 1;               // assignment returns 1
y = x * 7 / 2;              // y = ?     (try these out yourself
b = ( (x + y) >= 13 );      // b = ?      after learning about
x += -5;                    // x = ?      operators and
mult *= x - y;              // mult = ?   priority in the next
b = mult != 1337;           // b = ?      slides)
```

3. Basic statements: if, switch, loops

3.1. Statements and assignments:

Operators that we will use in this course:

Arithmetic operators:	`+`	`-`	`*`	`/`	`%`	`++`	`--`
Relational operators:	`>`	`>=`	`==`	`!=`	`<=`	`<`	`? :`
Logical operators:	`&&`	`||`	`!`				
Assignment operators:	`=`	`+=`	`-=`	`*=`	`/=`	`%=`	
	`<<=`	`>>=`					
Three-way comparison operator:	`<=>`						

unary operators
binary operators
unary and binary operators
ternary operator

3.1. Statements and assignments:

Arithmetic operators (try out yourself):

```
int x=5, y=9, width, length, multiply, division, remainder;

width     = x + y;                  // addition:       14
length    = width - 1;              // subtraction:    13
multiply  = x * y;                  // multiplication: 45
division  = x / y;                  // division:       0  (why?)
remainder = x % y;                  // modulo (remainder after division): 5
x++;                                // increment: x = x + 1:  6
y--;                                // decrement: x = x - 1:  8
length = width = y;                 // an assignment returns a value: 8
multiply = (x = 2) * (y = 5);       // (bad style) : 10
```

3.1. Statements and assignments:

Import to note:

- (Arithmetic) Operators do not exist for each type

  for example, `%` (modulo) does not exist for **`float`** or **`double`**

- Operators might have different behavior between types

```
int x = 7, y = 3, z;
z = x / y;       // note: z will get the value 2, since the
                 // operator '/' operates on two integers !
```

3.1. Statements and assignments:

Arithmetic operators, a first example (difficulty level: 🌶️)

```
/**
  Write a program that initializes two integers 'number1' and 'number2',
  and tests whether number1 is a multiple of number2. Put this result in the
  boolean variable 'result' below.
  */

int main() {
  bool result = false;



  return (int) result;
}
```

3.1. Statements and assignments:

Relational and logical operators (try out yourself):

```
double f1 = 15.2, f2 = 31.6;
bool b1 = true, b2 = false;
std::cout << (f1 == f1);                    // true  -> 1
std::cout << (b1 != b1);                    // false -> 0
std::cout << (b1 == true);                  // true  -> 1
std::cout << (f1 < f2);                     // true  -> 1
std::cout << (f1 <= f1);                    // true  -> 1
std::cout << ((f1 > f1) && (!b1));          // false -> 0
std::cout << ((b1 != b2) || (f2 >= f1) );   // true  -> 1
std::cout << (b1 || b2);                    // true  -> 1
std::cout << ( (f1 > 10.0) ? 'y' : 'n' );   // y
```

3.1. Statements and assignments:

Assignment operators (try out yourself):

```
float x = 5.1f, y = 9.2f;
int z = 7;

x = y + 1.0f;                    // results in x having the value 10.2f
y -= x;                          // y = y - x
x += 1.0f;                       // x = x + 1.0
y /= 2.0f;                       // y = y / 2.0
z %= 3;                          // z = z % 3 (note z is integer)
```

3.1. Statements and assignments:

Three-way comparison operator or spaceship operator <=> (C++20) returns an *object* that can be directly compared with a positive, a 0, or negative integer:

```
(3 <=> 5) == 0;       // false
('a' <=> 'a') == 0;   // true
(3 <=> 7) < 0;        // true
(7 <=> 5) < 0;        // false
```

3.1. Statements and assignments:

example 00 (difficulty level: 🌶️🌶️):

```
/**
 Try to figure out what is happening in the code below. Then try to compile the
 code, and then change the commented part so that the code's output becomes 1:
*/
#include <iostream>  // to allow use of std::cout and std::endl
int main() {
  auto i = 7;    // what type is variable i?
  auto j = 9.0;  // what type is variable j?

  bool ret = ( ( i <=> j) < 0 );  // change the '== 0' part so that the output is 1

  std::cout << ret << std::endl;
  return 0;
}
```

3.2. Priority

- Operators are not always executed from left to right, as statements
- **Use braces** to avoid having to learn the table below by heart:

Prio.	Operators	Associativity
11	`,`	left to right
10	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `? :`	right to left
9	`||`	left to right
8	`&&`	left to right
7	`<`, `>`, `<=`, `>=`	left to right
6	`<<`, `>>`	left to right
5	`+`, `-`	left to right
4	`*`, `/`, `%`	left to right
3	prefix `++`, prefix `--`, unary `-`, unary +, `!`, `(`type`)`, `new`, `delete`	right to left
2	suffix `++`, suffix `--`, `.`, ->,	left to right
1	`::`	left to right

3.2. Priority

- Examples:

```
int a = 2, b = 3, c = 4;
a = b * c + d;      // a << ((b * c) + d)
a /= b * c % d;     // a / ((b * c) % d)
a += b = c + d;     // a += (b = c + d)
```

Beware of prefix and postfix increment / decrement operators:

```
a = 10;
b = ++a;
// a = 11, b = 11
```

```
a = 10;
b = a++;
// a = 11, b = 10
```

3.3. Blocks

- Statement sequences and are executed one by one, from left to right:
  `length = 10; width = 15; surface = length * width;`
- A block of statements thus implements a function (e.g., between the curly braces `{` and `}` for main) and can be used anywhere where a statement can appear. But beware that variables defined there only 'live' in the block:

```
int main() {
  int length, width, surface;
  {
    length = 10; width = 15;
    surface = length * width;
  }
  return 0;
}
```

```
int main() {
  int length = 10, width = 15;
  {
    int surf = length * width;
  }
  return surf; // error: surf unknown
}
```

3.4. Selection statements: `if` and `switch`

- Depending on a condition, selection statements can either execute the next statement, or not
- Simply for one case:

```
if ( number < 0 )   // if number is negative
  sign = '-';       // then the sign is '-'
```

- For both cases:

```
if ( number < 0 )   // if number is negative
  sign = '-';       //  then the sign is '-'
else                // otherwise:
  sign = '+';       //  the sign is '+'
```

3.4. Selection statements: `if` and `switch`

Conditions often use relational operators:

```
int x, y, sum, counter, minimum;

if (x > y) ...                  // is x bigger than y?
if (x >= y) ...                 // is x bigger or equal than y?
if (sum == x + y) ...           // is sum equal to (x+y) ?
if (x != y) ...                 // is x different from y?
if (counter < 100) ...          // is counter smaller than 100?
if (x + 1 <= 1 - y) ...         // is (x+1) smaller or equal to (1-y)?

minimum = (x < y)? x : y;  // minimum gets a new value of ...
                           // if ( x < y ), then x, else y
```

3.4. Selection statements: `if` and `switch`

Conditions also often use logical operators:

```
int x, y, counter;
bool readFlag, writeFlag;

if (readFlag || writeFlag) ...    // logical OR: readFlag or
                                  // writeFlag need to be true,
                                  // will be skipped if both are false

if ((x != 0) && (y / x < 5)) ...  // logical AND: both y/x needs to be
                                  // smaller than 5 and x shouldn't be 0

if (!(x < 0)) ...   // NOT: the same as if (x >= 0)
```

3.4. Selection statements: `if` and `switch`

Nesting `if` statements:

```
if (number == 0)
  sign = 0;
else
  if (number > 0)
     sign = +1;
  else
     sign = -1;
```

Alternatively, using the `( ? : )` operator:

```
sign =  (number == 0) ? 0 : ( (number > 0) ? +1 : -1 );
```

3.4. Selection statements: `if` and `switch`

Nesting `if` statements can be tricky, **use curly braces**

```
if ( x == y )
  if ( y == z )
     allEqual = true;
else
     allEqual = false;
```

≠

```
if ( x == y ) {
  if ( y == z )
     allEqual = true;
}
else
     allEqual = false;
```

=

```
if ( x == y )
  if ( y == z )
     allEqual = true;
  else
     allEqual = false;
```

=

```
if ( x == y ) {
  if ( y == z )
     allEqual = true;
  else
     allEqual = false;
}
```

3.4. Selection statements: `if` and `switch`

Nesting many **`if`** statements:

```
if (menuItem == 1) {
          ...
} else if (menuItem == 2) {
          ...
} else if ((menuItem == 3) ||
           (menuItem == 4)) {
          ...
} else if (menuItem == 5) {
          ...
} else {
          ...
}
```

Using one **`switch`** statement:

```
switch (menuItem) {
  case 1:  ...
           break;
  case 2:  ...
           break;
  case 3:
  case 4:  ...
           break;
  case 5:  ...
           break;
  default: ...
           break;
}
```

3.5. Loops

3.5.1. The `while` loop:

```
int i, sum;
sum = 0;
i = 1;
while ( i < 7 ) {  // this block
  sum += i;        // is executed
  i++;             // repeatedly
}
```

result:

sum: `0, 1, 3, 5, 9, 14, 20`

3.5.2. The `do while` loop:

```
int i, sum;
sum = 0;
i = 1;
do {                // this block
  sum += i;         // is executed
  i++;              // repeatedly

} while ( i < 7 );
```

result:

sum: `0, 1, 3, 5, 9, 14, 20`

3.5. Loops

3.5.3. The `for` loop:

```
int i, sum = 0;
for ( i = 0; i <= 10; i++ ) {  // this block
  sum += i;                    // is executed repeatedly
}
```

Template:

initialization of loop variable

do we loop again?

in- or decrease the loop variable

```
for ( statement ; condition ; statement )
    <statement> ;
```

statement, or block of statements to repeat

3.5. Loops: **break** and **continue**

- **break** terminates a loop. The rest of the loop body will not be executed:

```
int num = 10;
while (num > 0) {
    if (num == 5) break;  // stop after num has reached 5
    num--;
}
```

- **continue** does not terminate the loop, but just skip the rest of the loop body:

```
int num = 10;
while (num > 0) {
    if (num == 5) continue;  // when num == 5, don't decrement
    num--;
}
```

3.5. Loops: When do we use which loop type?

- **`for`** loop:
  - for a given range (e.g. ” for all i = 1 … N “)
  - when you automatically need a loop variable
- **`while`** loop:
  - when the (maximal) number of repetitions is not known beforehand
  - when the repetition conditions are more complex
- **`do while`** loop:
  - when a block needs to be repeated at least once

3.5. Loops: example 02 (difficulty level: 🌶️)

```
/**
  Write a program that prints out a series of numbers, starting at 120.0 and where
  each next number is seven less than the previous one. Stop once the number is
  smaller than 43.7
  */
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {




  return 0;
}
```

3.5. Loops: example 03 (difficulty level: 🌶🌶)

```
/**
  Write a program that asks the user for a number, and then prints out this number
  in the terminal, followed by the half of the previous number until
  the result is smaller than ten. So for 100 it would give out: 100, 50, 25.5, 12.25
  */
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {




  return 0;
}
```

3.5. Loops: example 04 (difficulty level: 🌶🌶)

```
/**
  Write a program that counts from 131 down till 23, one number per line in the
  the terminal, and prints out "hop", if the number is a multiple of 7.
  */
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {




  return 0;
}
```

3.5. Loops: example 05 (difficulty level: 🌶🌶🌶)

```
/**
 Write a program that prints in the terminal all prime numbers from 3 till 99.
 Remember: A number is a prime when any division by a smaller number results in
 a remainder that is never zero.
*/
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {




  return 0;
}
```

3.5. Loops: example 06 (difficulty level: 🌶🌶🌶)

```
/**
 Write a program that draws in the terminal a big X out of the character 'X',
 depending on the variable int size (with size = 3, 4, ..., 20):
 size = 3:     size = 4:      size = 5:      etc.
   X X          X  X           X   X
    X            XX             X X
   X X           XX              X
                X  X            X X
                                X   X      */
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {



  return 0;
}
```

3.5. Loops: example 07 (difficulty level: 🌶🌶🌶🌶)

```
/**
 Write a program that draws in the terminal a bigger X out of the character 'X',
 depending on the variable int size (with size = 3, 4, ..., 20):
 size = 3:     size = 4:      size = 5:      etc.
   XX X         XX  X          XX   X
    XX           XXX            XXXX
   X XX          XXX             XX
                X  XX           X XX
                               X   XX      */
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {



  return 0;
}
```

4. Functions, Recursion, Call by Value

4.1. Functions and their parameters
4.2. Recursive Functions
4.3. Call by Value
4.4. `inline` Functions, Overloading, `=delete`
4.5. Default Parameters and Function Attributes
4.6. Header files and Modules
4.7. Miscellaneous: Variadic arguments and Coroutines

4.1. Functions and their parameters

- Blocks of code can sometimes re-use the same variables and need to be used throughout a program
- For example calculating the maximum of two integers:

```
int maximum = 0, a = 12, b = 10;
{
  if (a > b) {
    maximum = a;
  } else {
    maximum = b;
  }
}
// maximum now holds the value of a or b, whichever is largest
```

4.1. Functions and their parameters: Declaring Functions

- Before you can use (call) a function, you have to declare it (similar to how we have to declare variables before use).
- A function declaration contains a return type, function name, and parameters, example:

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

- You typically declare *and* implement the function before **`main()`**, example:

```
int maximum( int a, int b ) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

4.1. Functions and their parameters: Declaring Functions

- With each function call, formal parameters need actual parameters, *unless* the function prototype has default values:

```
#include <iostream>  // output to the console
#include <cstdint>  // we're using the uint16_t type
void drawLine(char symbol = '-', uint16_t len = 25) {
  for (auto line = 0; line < len; line++) std::cout << symbol;
  std::cout << std::endl;
}
int main() {
  drawLine();  // writes 25 times the '-' symbol to console
  drawLine(50);  // writes 50 times the '-' symbol to console
  drawLine('=', 9);  // writes 9 times the '=' symbol to console
  return 0;
}
```

4.1. Functions and their parameters: Declaring Functions

- Functions can call other functions, allowing cycles:
  function **`a()`** calls **`b()`**, **`b()`** calls **`a()`**
  → In this case, declarations need to come first, example:

```
int a();  // declaration of function a()
int b();  // declaration of function b()
int a() {  // implementation of function a():
  std::cout << "Yes" << std::endl;
  return b();
}
int b() {  // implementation of function b():
  std::cout << "No"<< std::endl;
  return a();
}
```

4.1. Functions and their parameters: Declaring Functions

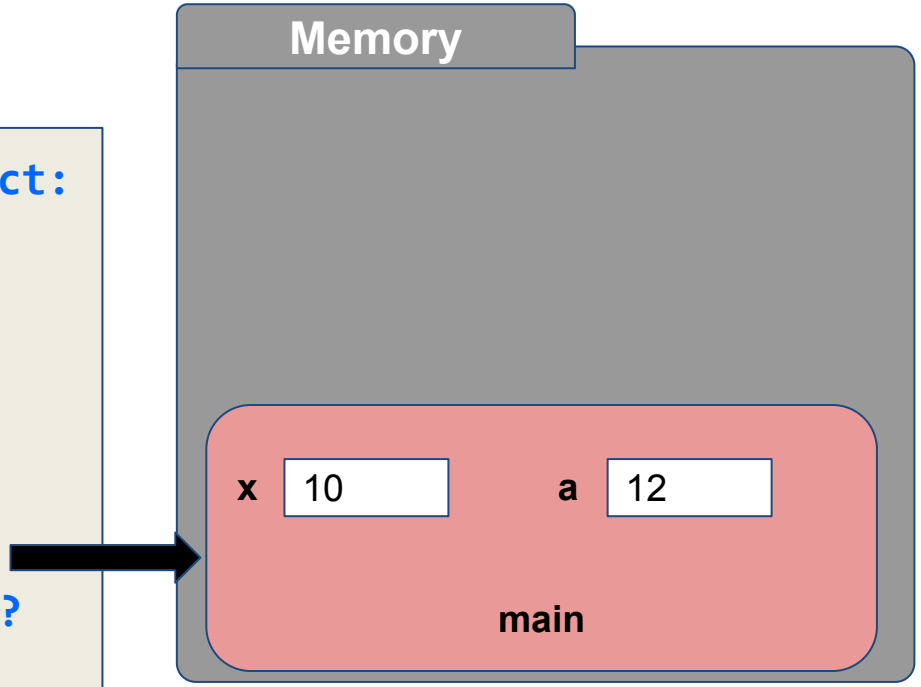
- A function declaration can have *parameters*: variables that obtain a value when the function is called and that are treated as local variables in the implementation of the function
- A function can have a return type. If not, we use `void` → Is this a type?

```
void printMaximum( int a, int b ) {  // a and b are parameters
  if (a > b) {                       // a and b can be used as variables of
    std::cout << a;                  // type integer in the implementation of
  } else {                           // the function
    std::cout << b;
  }
  std::cout << std::endl;   // note that we don't return anything
}
```

4.1. Functions and their parameters: Using Functions

- A function is *called*:

```
// declare & implement myFunct:
int myFunct(int b, int a) {
  a = 2 * b + a * a;
  return a + 1;
}
// now we can call myFunct:
int main() {
  int x = 10; int a = 12;
  a = myFunct(a, x+1);  // a?
  return 0;
}
```

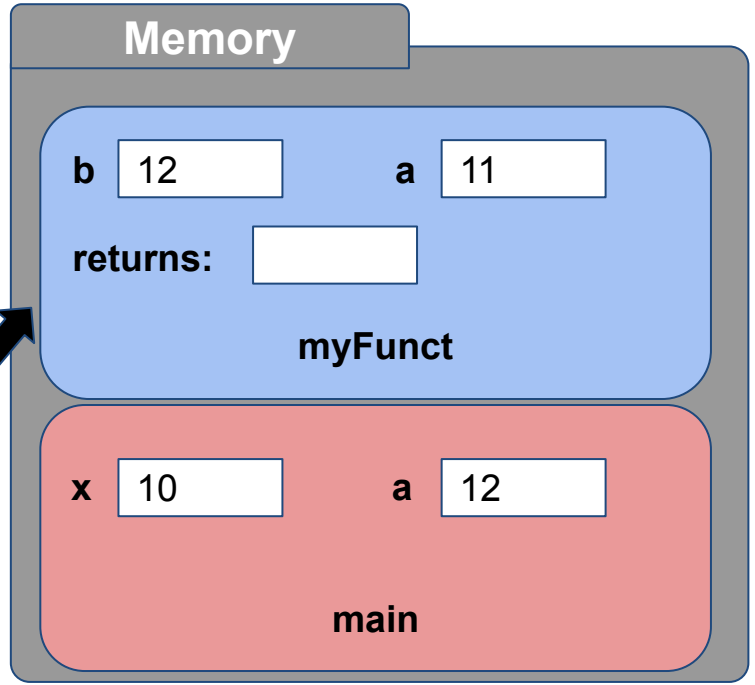


A stack is created in memory, in which the function's local variables are stored

4.1. Functions and their parameters: Using Functions

- A function is *called*:

```
// declare & implement myFunct:
int myFunct(int b, int a) {
  a = 2 * b + a * a;
  return a + 1;
}
// now we can call myFunct:
int main() {
  int x = 10; int a = 12;
  a = myFunct(a, x+1);  // a?
  return 0;
}
```

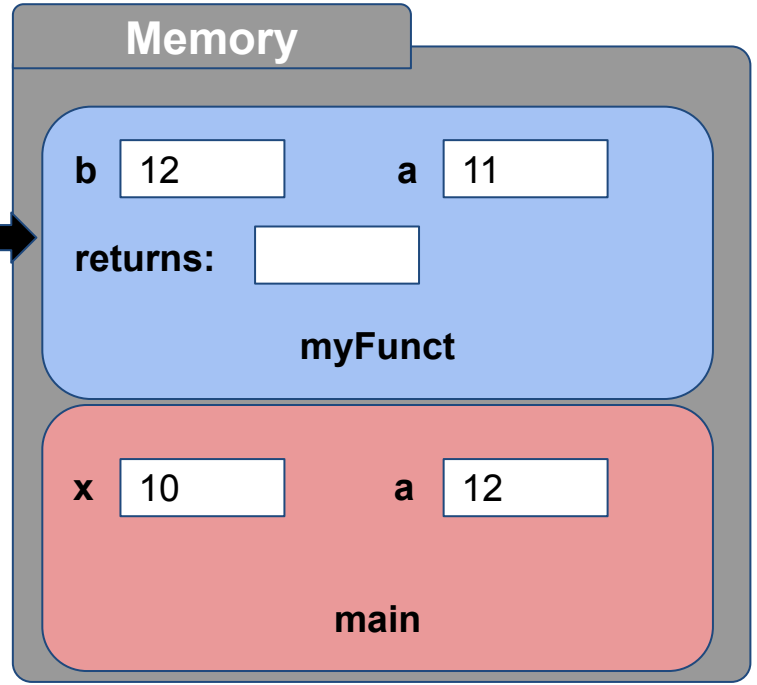


A stack is created in memory, in which the function's local variables are stored

4.1. Functions and their parameters: Using Functions

- A function is *called*:

```
// declare & implement myFunct:
int myFunct(int b, int a) {
  a = 2 * b + a * a;
  return a + 1;
}
// now we can call myFunct:
int main() {
  int x = 10; int a = 12;
  a = myFunct(a, x+1);  // a?
  return 0;
}
```

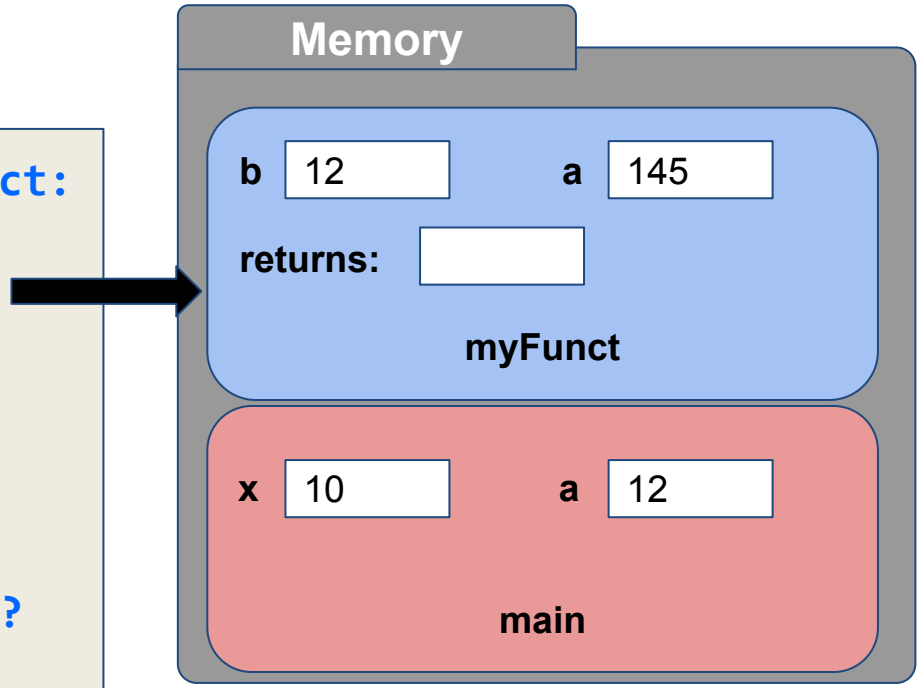


A stack is created in memory, in which the function's local variables are stored

4.1. Functions and their parameters: Using Functions

- A function is *called*:

```
// declare & implement myFunct:
int myFunct(int b, int a) {
  a = 2 * b + a * a;
  return a + 1;
}
// now we can call myFunct:
int main() {
  int x = 10; int a = 12;
  a = myFunct(a, x+1);  // a?
  return 0;
}
```

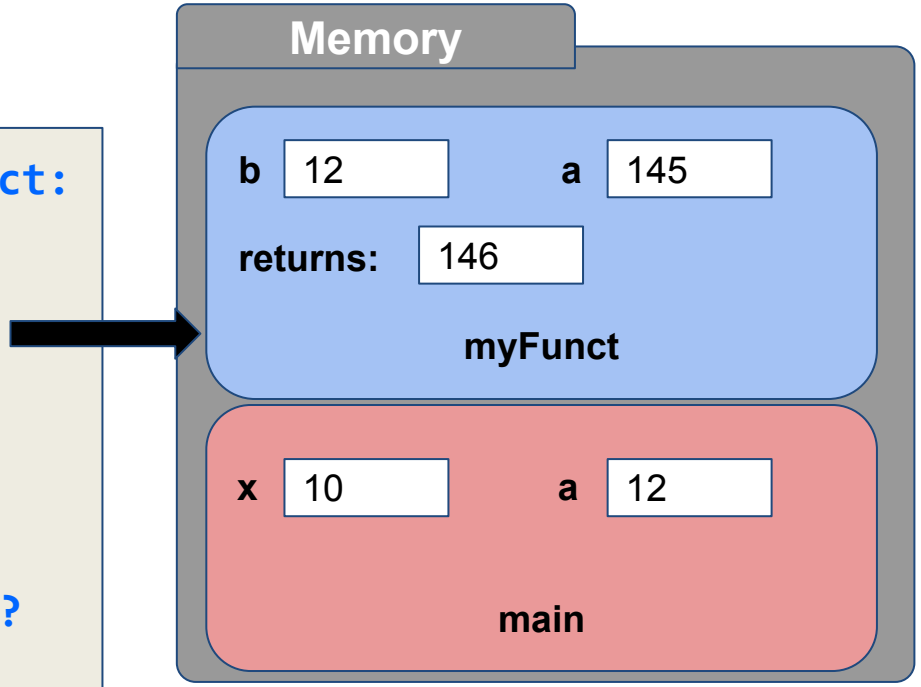


A stack is created in memory, in which the function's local variables are stored

4.1. Functions and their parameters: Using Functions

- A function is *called*:

```
// declare & implement myFunct:
int myFunct(int b, int a) {
  a = 2 * b + a * a;
  return a + 1;
}
// now we can call myFunct:
int main() {
  int x = 10; int a = 12;
  a = myFunct(a, x+1);  // a?
  return 0;
}
```

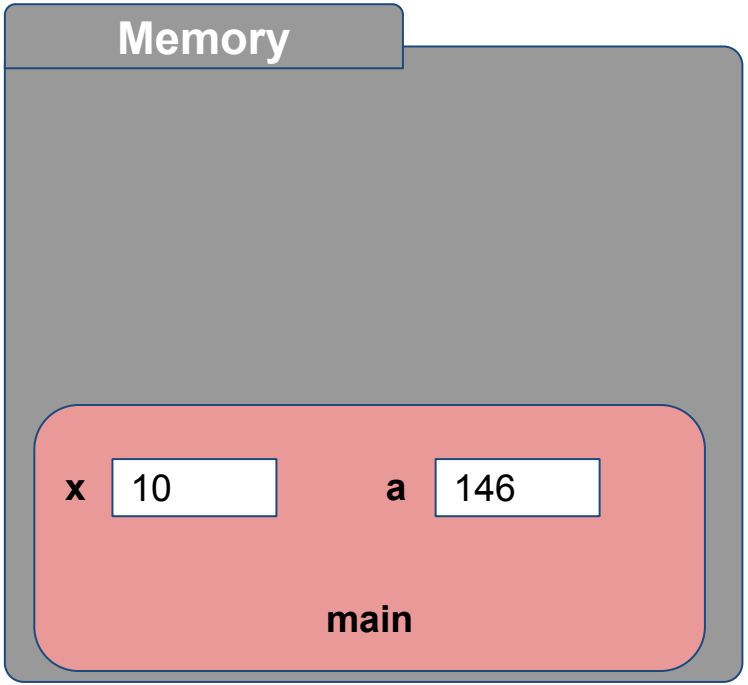


A stack is created in memory, in which the function's local variables are stored

4.1. Functions and their parameters: Using Functions

- A function is *called*:

```
// declare & implement myFunct:
int myFunct(int b, int a) {
  a = 2 * b + a * a;
  return a + 1;
}
// now we can call myFunct:
int main() {
  int x = 10; int a = 12;
  a = myFunct(a, x+1);  // a?
  return 0;
}
```



A stack is created in memory, in which the function's local variables are stored

4.1. Functions and their parameters: Using Functions

- Maze Game v.1.0: expand this code to move the player and add color

```
/* First draft of Maze Game: draw the player, respond to key presses */
#include <ncurses.h>  // functions to draw colored text in terminal
int main() {
  char c = ' ';          // used for user key input
  auto x = 10, y = 5;    // (x,y) position of player: start at (10,10)
  initscr(); curs_set(0);  // ncurses: initialize window, then hide cursor
  while ( c != 'q' ) {   // as long as the user doesn't press q ..
    mvaddch(y, x, '@');  // ncurses function: draw a @ at position (x,y)
    c = getch();         // capture the user's pressed key
    // handle here the moving
  }
  endwin();            // ncurses function: close the ncurses window
  return 0;
}
```

4.1. Functions and their parameters: Using Functions

```
/* First draft of Maze Game: draw the player, respond to key presses
   Result of the in-class programming code (see YouTube video of the lecture)
*/

#include <ncurses.h>  // functions to draw colored text in terminal

// initialize all the functions to start drawing in ncurses
void initNCurses() {
  initscr(); curs_set(0);  // ncurses: initialize window, then hide cursor
  noecho();  // don't show keys pressed in terminal
  start_color();  // use color
  init_pair(1, COLOR_BLUE, COLOR_GREEN);
  init_pair(2, COLOR_RED, COLOR_YELLOW);
}
```

4.1. Functions and their parameters: Using Functions

```
void clearScreen() {
  attron(COLOR_PAIR(1));  // set color pair to 1
  for ( auto line = 0; line < LINES; line++) {
    for ( auto col = 0; col < COLS; col++) {
      mvaddch(line, col, '.');  // ncurses function: draw '.' at (x,y)
    }
  }
  attroff(COLOR_PAIR(1));
}

// draw a symbol at (x,y) with color colorpair
void draw(int x, int y, char symbol, int colorpair) {
  attron(COLOR_PAIR(colorpair));  // set color pair to 1
  mvaddch(y, x, symbol);  // ncurses function: draw '.' at (x,y)
  attroff(COLOR_PAIR(colorpair));
}
```

4.1. Functions and their parameters: Using Functions

```
int main() {
  auto c = ' ';          // used for user key input
  auto x = 10, y = 10;   // (x,y) position of player: start at (10,10)
  initNCurses();         // initialize ncurses functionality
  while ( c != 'q' ) {    // as long as the user doesn't press q ..
    clearScreen();
    draw(x, y, '@', 2);  // draw our player
    c = getch();          // capture the user's pressed key
    switch (c) {
      case 'w': y--; break;  // go up
      case 's': y++; break;  // go down
      case 'a': x--; break;  // go left
      case 'd': x++; break;  // go right
    }
  }
  endwin();           // ncurses function: close the ncurses window
  return 0;
}
```

4.2. Recursive Functions

- A function can call itself. For example in a function to calculate the factorial of a number (notation: n!)

```
// factorial of n (n!):
double factr(double n) {
  if (n == 0.0)
    return 1.0;
  else if (n > 0.0)
    return n * factr(n-1);
}
```

```
double f = factr(3.0);
```

Mathematical definition:

$$n! = \begin{cases} 1 & \text{for } n = 0 \\ n \cdot (n-1)! & \text{for } n > 0 \end{cases}$$

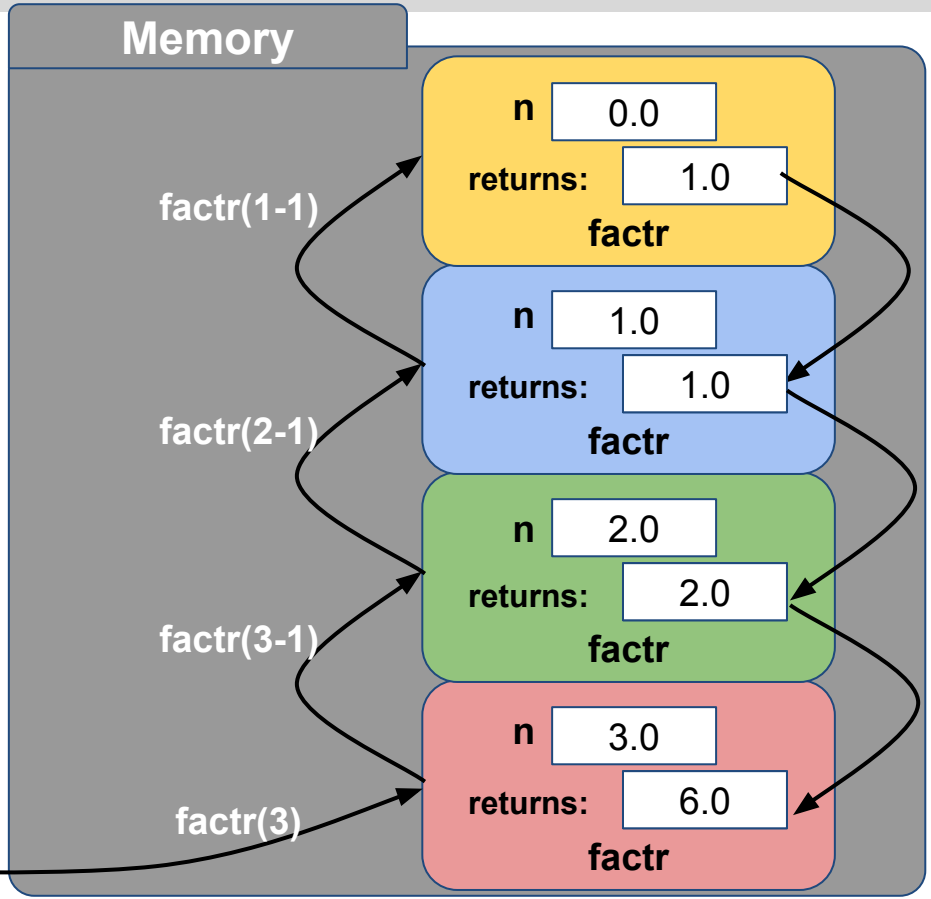
so:
2! = 2 . 1 = 2
3! = 3 . 2 . 1 = 6
4! = 4 . 3 . 2 . 1 = 24
5! = 5 . 4 . 3 . 2 . 1 = 120
and so on …

4.2. Recursive Functions

- Whenever a function is called, a new space is reserved in memory for parameters and local variables. Example:

```
double factr(double n) {
  if (n == 0.0)
    return 1.0;
  else if (n > 0.0)
    return n * factr(n-1);
}
```

```
double f = factr(3.0);
```



4.3. Call by Value

In C++, most parameters are passed **by value**

- This means, a function always receives **copies** of the actual parameters
- When the function is called, the values of the actual parameters are assigned to the formal parameters in the function declaration:

```
double factr(double n);  // n is a formal parameter of factrr
```

```
double y = factr(6.0);   // 6.0 is the actual parameter of factr
```

- With call-by-value, variables given as actual parameters are never changed
- The same variable can be simultaneously passed to multiple parameters:

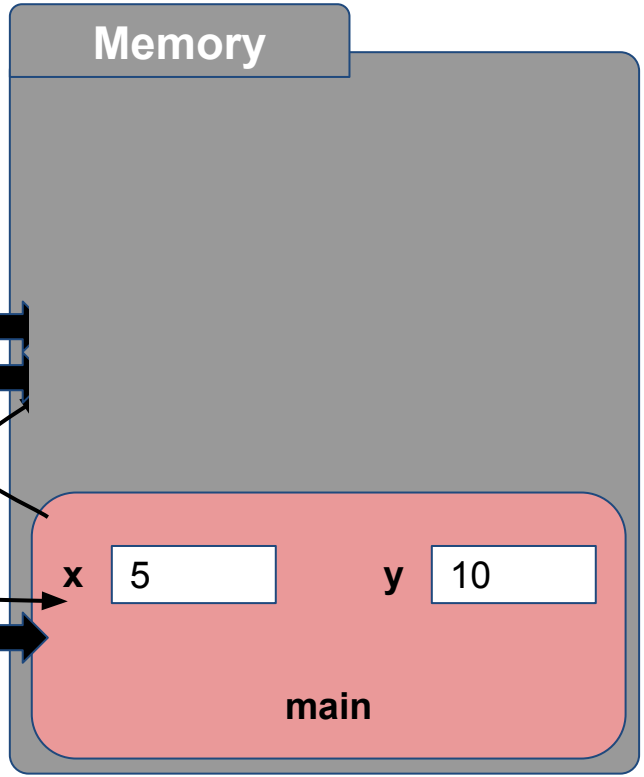
```
int a = 10;
y = maximum(a, a);    // the value 10 is copied to both parameters
```

4.3. Call by Value

In C++, parameters are passed **by value**

So the variable does not get passed, *just its value*

```
#include <iostream>  // output to terminal
void swap(int x, int y){
  int temp = 0;
  temp = x; x = y; y = temp;
}
int main()  {
  int x = 5, y = 10;
  swap(x, y);
  std::cout << x << ", " << y << std::endl;
  return 0;
}
```

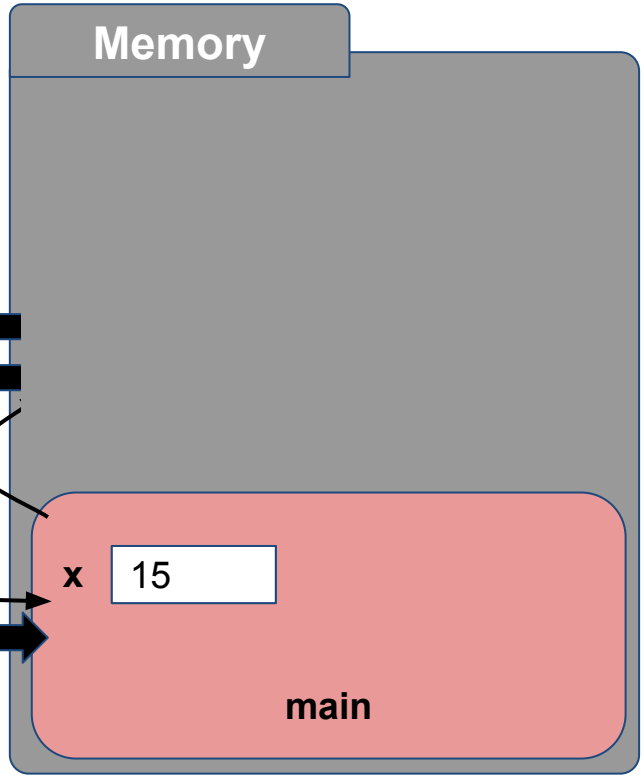


4.3. Call by Value

In C++, parameters are passed **by value**

You can use the function's return value:

```
#include <iostream>  // output to terminal
int addFive(int x) {
  x += 5;
  return x;
}
int main()  {
  int x = 10;
  x = addFive(x);
  std::cout << x << std::endl;
  return 0;
}
```



4.4. `inline` Functions, Overloading, `=delete`

- **`inline`** tells the compiler that inline substitution of a function is preferred over function call: instead of calling the function and transferring control to the function body, a copy of the function body is executed
- This avoids overhead from the function call (passing the arguments and retrieving the result)
- This may result in a larger executable (due to repeating multiple times)

```
inline int maximum( int a, int b ) {
  return (a > b)? a : b;
}
```

4.4. `inline` Functions, Overloading, `=delete`

- Sometimes, the same functionality is needed on different types:

```
auto maximum( int a, int b );
auto maximum( double a, double b );
auto maximum( char a, char b );
```

(note that `auto` is not allowed for the function's parameters, deduced return types are a C++14 extension)

- Multiple functions with the same name are allowed, if
  - the number of parameters are different, or
  - at least one parameter has a different type
- This is ***overloading*** the function name, and should be used for multiple functions of the same functionality. Note that with subtle differences, like signed/unsigned, float/double, it is hard to predict what will be called

4.4. `inline` Functions, Overloading, `=delete`

- There are four Overloading Resolution Rules
  - An exact match between parameter types
  - A promotion (e.g., char to int)
  - A standard type conversion (e.g. float and int)
  - A constructor or user-defined type conversion (see later)

- `= delete` can be used to prevent calling the wrong overload:

```
void myFunction(int) { ; }
void myFunction(double) = delete;
int main() {
  myFunction(7);    // this is fine
  myFunction(7.0);  // this results in a compilation error
  return 0;
}
```

4.5. Default Parameters and Function Attributes

- Parameters can be given a default value (If the call does not supply a value for this parameter, this default value will be used):
  - All default parameters must be the *rightmost* parameters
  - Default parameters must be declared only once
  - Default parameters can improve compile time and avoid redundant code because they avoid defining other overloaded functions

```
void myFunction(int a, int b = 7);   // declaration of myFunction

void myFunction(int a, int b) { ; }  // definition of myFunction
int main() {
  myFunction(8);    // this is fine, a = 8, b = 7
  return 0;
}
```

4.5. Default Parameters and Function Attributes

- Functions can be marked with standard properties, to express their intent:
  - `[[noreturn]]` indicates that a function does not return, for optimization purposes or compiler warnings (from C++11)
  - `[[deprecated]]` , `[[deprecated("reason")]]` indicates that the use of a function is discouraged through a compiler warning (from C++14)
  - `[[nodiscard]]` , `[[nodiscard("reason")]]` (C++17, resp. C++20) throws a warning if the function's return value is not handled

```
[[noreturn]] void myFunction() { std::exit(0); }

[[deprecated("old function, use newFunction instead")]]
void oldFunction(int p) { ... }

[[nodiscard("please handle return value")]] int getMaximum() { ... }
```

4.6. Header files and Modules

- It is likely that any code you will write will have to be split into several functions that call each other, instead of implementing everything in the `main()` function
- We define and implement these functions in separate files, if they form a collection that belong to each other (see for example the functions we used from ncurses)
- This is a **module**: a part of a program that can be compiled separately
- In C++, a module always should consist of two files:
  - a **header** file (`*.h`), which contains the function declarations
  - an **implementation** file (`*.cpp`), in which the functions are implemented

4.6. Header files and Modules

Maze.cpp

```
/* Second draft of Maze Game: drawing functions are our module "drawMaze" */
#include "drawMaze.h"  // functions related to drawing
int main() {
  auto c = ' ';          // used for user key input
  auto x = 10, y = 10;   // (x,y) position of player: start at (10,10)
  initNCurses();         // initialize ncurses functionality
  while ( c != 'q' ) {   // as long as the user doesn't press q ..
    clearScreen();
    draw(x, y, '@', 2);   // draw our player
    c = getch();          // capture the user's pressed key
    switch (c) {
      case 'w': y--; break;  // go up
      case 's': y++; break;  // go down
      case 'a': x--; break;  // go left
      case 'd': x++; break;  // go right
    }
  }
  endwin();           // ncurses function: close the ncurses window
  return 0;
}
```

4.6. Header files and Modules

drawMaze.h

```
/* Drawing functions declared */
#include <ncurses.h>  // functions to draw colored text in terminal

// initialize all the functions to start drawing in ncurses and use color
void initNCurses();

// clear the screen
void clearScreen();

// draw a symbol at (x,y) with color colorpair
void draw(int x, int y, char symbol, int colorpair);
```

4.6. Header files and Modules

drawMaze.cpp

```
/* Drawing functions implemented */
#include "drawMaze.h"  // functions to draw colored text in terminal

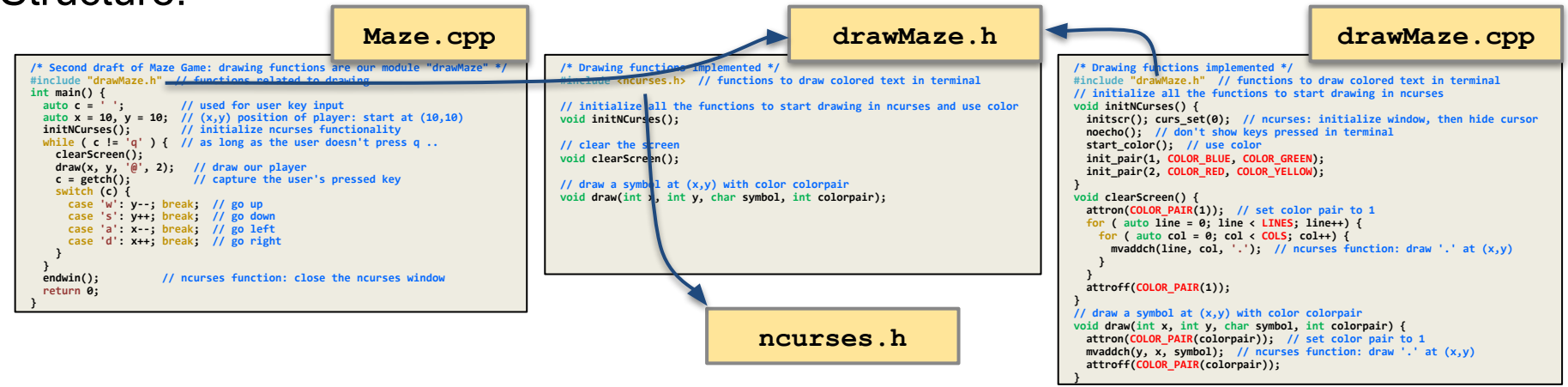
// initialize all the functions to start drawing in ncurses
void initNCurses() {
  initscr(); curs_set(0);  // ncurses: initialize window, then hide cursor
  noecho();  // don't show keys pressed in terminal
  start_color();  // use color
  init_pair(1, COLOR_BLUE, COLOR_GREEN);
  init_pair(2, COLOR_RED, COLOR_YELLOW);
}
void clearScreen() {
  attron(COLOR_PAIR(1));  // set color pair to 1
  for ( auto line = 0; line < LINES; line++) {
    for ( auto col = 0; col < COLS; col++) {
      mvaddch(line, col, '.');  // ncurses function: draw '.' at (x,y)
    }
  }
  attroff(COLOR_PAIR(1));
}
```

4.6. Header files and Modules

drawMaze.cpp

```
// draw a symbol at (x,y) with color colorpair
void draw(int x, int y, char symbol, int colorpair) {
  attron(COLOR_PAIR(colorpair));  // set color pair to 1
  mvaddch(y, x, symbol);  // ncurses function: draw '.' at (x,y)
  attroff(COLOR_PAIR(colorpair));
}
```

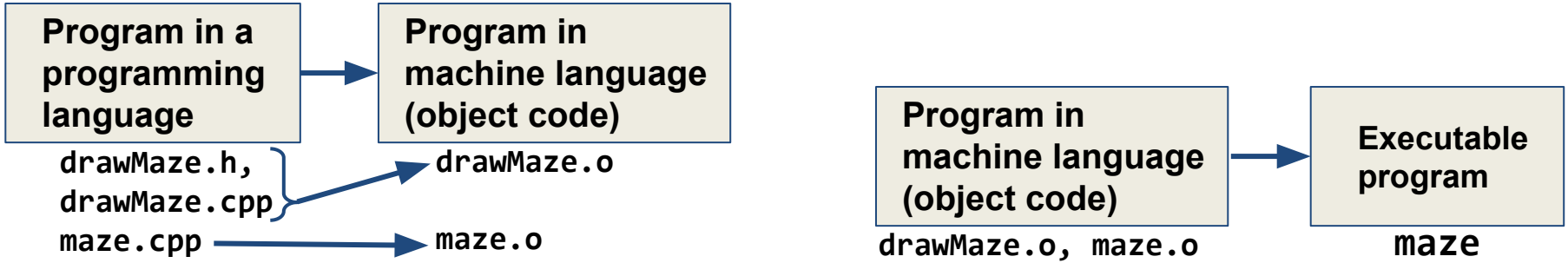
Structure:



4.6. Header files and Modules

Maze Game v.2.0: How to compile the program?

- First compile the module and the program into object files:
  `g++ -c drawMaze.cpp -std=c++11` → object file `drawMaze.o`
  `g++ -c maze.cpp -std=c++11` → object file `maze.o` is created
- Then link the object files:
  `g++ maze.o drawMaze.o -o maze -l ncurses`

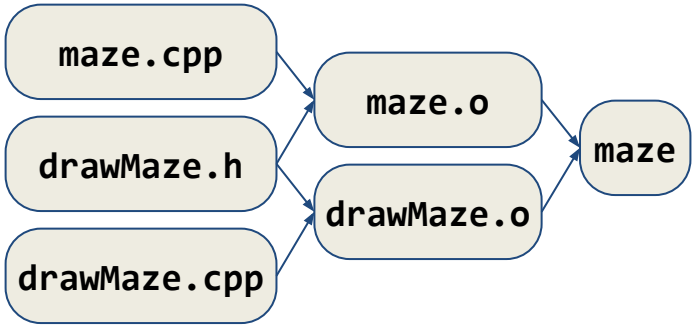


4.6. Header files and Modules

- Why use modules?
  - To **better structure** the program code: Separate modules make it easier to divide your code and find where you need to change or continue your source code
  - Make modules **re-usable** by others: Anyone can read the header (`*.h`) file and will know what functions they can use if the module is included, reading the implementation (`*.cpp`) is not needed
  - **Save compilation time**: Object files are already compiled, they just need to be linked to other modules and the program code

4.6. Header files and Modules: The **make** utility

- Revisiting the Maze Game v.2.0, we have these *dependencies*:



compile **drawMaze.cpp:**

```
g++ -c drawMaze.cpp -std=c++11
```

compile **maze.cpp:**

```
g++ -c maze.cpp -std=c++11
```

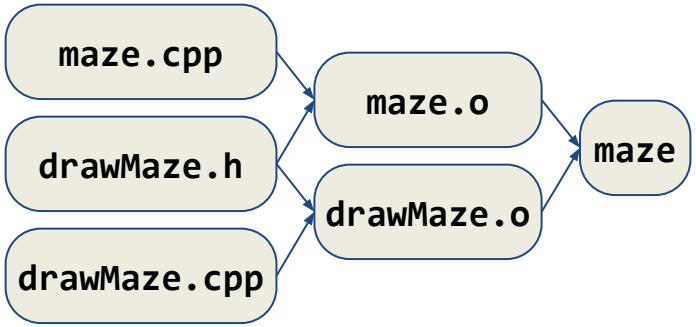
link the objects files into the executable program **maze:**

```
g++ maze.o drawMaze.o -o maze -l ncurses
```

- After a change, we want to recompile only the affected files
- The **make** program automates this process for us:
  just type **make** in the terminal, in the code's directory

4.6. Header files and Modules: The **make** utility

- We need to tell **make** about these dependencies in a specific file that we need to create in the code's directory: **Makefile**



- After each rule, we need to type a **tab** before each g++ command in **Makefile**

Makefile

```
# Rule to make our program when
# 'drawMaze.o' and 'maze.o' are compiled:
maze: drawMaze.o maze.o
  g++ drawMaze.o maze.o -o maze -l ncurses
# Rule for dependency 'maze.o':
maze.o: maze.cpp drawMaze.h
  g++ -c maze.cpp -std=c++11
# Rule for dependency 'drawMaze.o':
drawMaze.o: drawMaze.cpp drawMaze.h
  g++ -c drawMaze.cpp -std=c++11
```

Summary

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

- A function returns at most one value and thus must have a return type (either `int`, `float`, `double`, `bool`, `char`, etc., or `void`: no return value)
- A function has a name and a list of parameters between braces
- The parameters are variables of the types `int`, `float`, `double`, `bool`, `char`
- The function is implemented as a block following the function definition, between curly braces:
- Each time this function is called, these statements are executed with any parameters as local variables

```
int maximum( int a, int b ) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

5. Arrays, Multidimensional Arrays

5.1. Arrays: Reminders

Types (`int`, `float`, `double`, `bool`, `char`, etc.) tell the compiler:

- the size of the variables (e.g., 4, 8, 1 bytes) in memory
- how these bits in memory should be interpreted
- and know the possible operations on them

For example:

if **`height`** and **`width`** are variables of type `int`, then the compiler knows

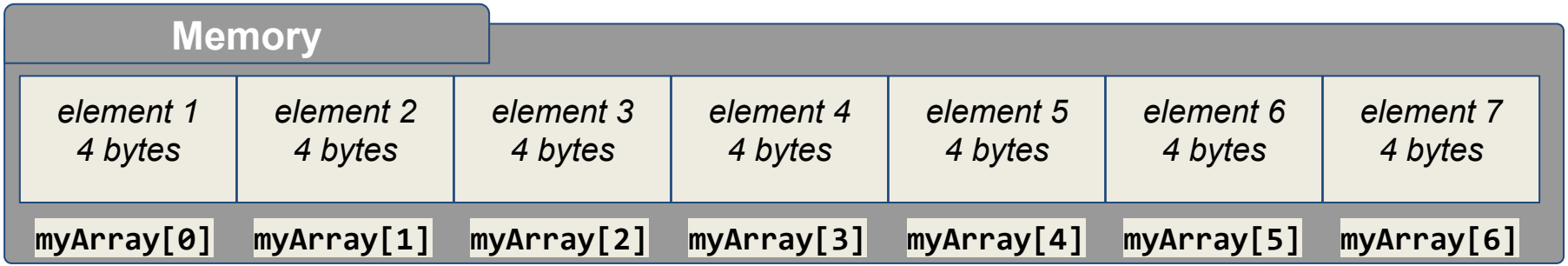
- that 4 bytes need to be reserved for each of them,
- which are organized so they span the whole numbers from -2147483648 to 2147483647
- and that **`height`** * **`width`** is a legal operation

5.1. Arrays

- An array is a serially numbered collection of variables that are all of the same *type*
- The number of elements is the *size* of the array
- Array elements are accessible via their *index*, from 0 to size-1

For example:

`float myArray[7];` is an array of 7 `float` variables, indexed from 0 to 6:



5.1. Arrays: Initialization, sizeof

- An array can be initialized by listing the elements between curly braces, **{** and **}**, and separated by commas:

```
double myArray[] = {1.09, 2.18, 4.36, 8.72};
```

  In this case, the array will automatically get the size 4

- `sizeof` is built-in operator that returns the number of *bytes* for the given variable or type:

```
int myArraySize = sizeof(myArray) / sizeof(myArray[0]);  // 16/4
```

- Loops are typically used for larger arrays:

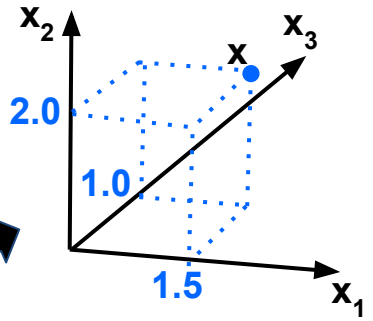
```
bool myArray[400];
for (int i = 0; i < 400; i++) myArray[i] = false;
```

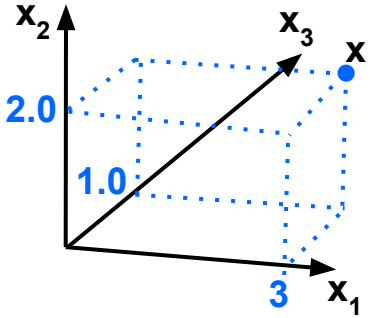
5.1. Arrays

- Example: a three-dimensional vector

```
double y[3]; // y is a 3d vector
y[0] = 1.5;
y[1] = 2.0;
y[2] = 1.0;
// or shorter:
double x[] = { 1.5, 2.0, 1.0 };

x[0] = 3.0;
```



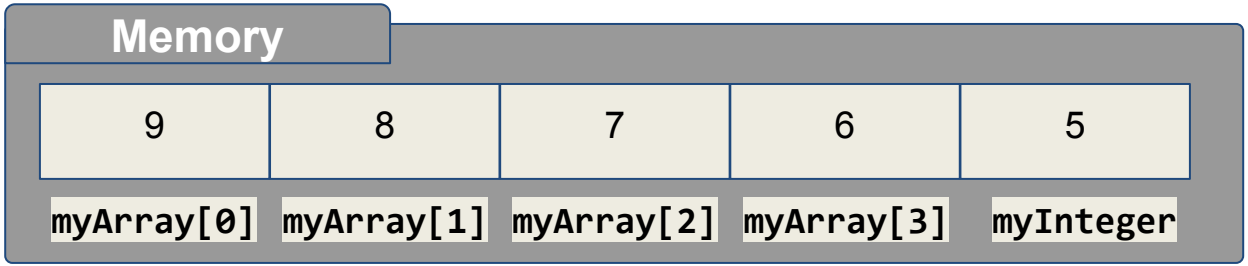


5.1. Arrays: Writing beyond the array boundary

- Most C++ compilers allow using *any* array indices to access array elements, even incorrect ones
- Non-existing array elements are usually other parts of memory, such as other variables or program code:

```
int myArray[4] = {9, 8, 7, 6};
int myInteger = 5;
std::cout << myArray[4] << std::endl;  // returns only a warning
```

- What could happen: `myArray[4]` returns the value of myInteger:



5.1. Arrays

Example 01 (difficulty level: 🌶️)

```
/**
  Write a program that initializes an array of 50 booleans, repeatedly having two
  elements with a true value, followed by one element with false.
  So the array starts with: true, true, false, true, true, false, true, true, …
  Do not use any variables other than myArray.
*/

int main() {
  bool myArray[50];



  return 0;
}
```

5.1. Arrays

Example 02 (difficulty level: 🌶️🌶️)

```
/**
  Write a program that lets a user fill an array of 10 integers, using a loop,
  and then calculate and output the average of all given numbers to the terminal.
  Assume that the user enters a valid number each time.
  */
#include <iostream>  // to allow use of std::cout, std::cin, and std::endl
int main() {
  int myArray[10];




  return 0;
}
```

5.1. Arrays

Example 03 (difficulty level: 🌶🌶🌶)

```
/**
  Write a program that draws a histogram or bar chart through
  an array of 17 integers. To 'draw' the bars, use the string
  "\u2589" or an empty space.
  */
#include <iostream>  // std::cout, std::cin, and std::endl
#include <random>    // rand(), returns a pseudo-random int
int main() {
  int myArray[17];
  for (int i = 0; i < 17; i++) {  // fill array with random
    myArray[i] = ( rand() % 25 ); // numbers between 0 and 24
    // draw here with std::cout and std::endl
    std::cout << myArray[i] << std::endl;
  }
  return 0;
}
```

example output:

```
7 ▉▉▉▉▉▉▉
24▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
23▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
8 ▉▉▉▉▉▉▉▉
5 ▉▉▉▉▉
22▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
19▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
3 ▉▉▉
23▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
9 ▉▉▉▉▉▉▉▉▉
15▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
15▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
17▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
17▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
12▉▉▉▉▉▉▉▉▉▉▉▉
3 ▉▉▉
2 ▉▉
```

5.2. Multidimensional Arrays

- An array can be multidimensional, for example 2-dimensional:
  ```
  int myTable[2][4] = { {1, 2, 3, 4}, {5, 6, 7, 8} };
  ```
- This array is essentially an array of 2 arrays: `myTable[0]`, `myTable[1]`
- Initialization of larger arrays typically needs nested loops:
  ```
  double map[100][20];
  for (int x = 0; x < 100; x++) {
    for (int y = 0; y < 20; y++) {
      map[x][y] = 0.0;
    }
  }
  ```
- `sizeof(myTable)` will return the total size, so 2x4x4=32 bytes
- `sizeof(myTable[0])` will return 4x4 = 16 bytes

5.2. Multidimensional Arrays: Maze Game v.3.00

- Expand on version 2.00 by drawing an actual maze in the screen background, in a tiled way (since the screen can be any size)
- Add this as a two-dimensional array that you initialize yourself in the **`redraw`** function to build up a maze, for example:

```
auto maze[][15] = { {1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1},
                    {0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
                    {1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0},
                    {1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0},
                    {1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1},
                    {1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1},
                    {0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1},
                    {1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0},
                    {1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0},
                    {1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0}
                  };  // array for drawing a maze as a background
```

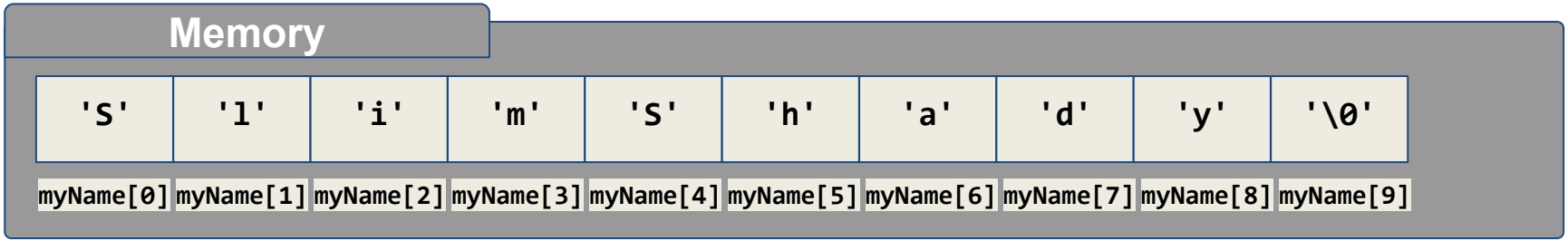
5.2. Multidimensional Arrays: Maze Game v.3.00

```
/* Third draft of Maze Game: We add an actual maze to our module "drawMaze" */
#include "drawMaze.h"  // functions related to drawing the maze and player
int main() {
  auto c = ' ';          // used for user key input
  auto x = 10, y = 10;   // (x,y) position of player: start at (10,10)
  initNCurses();         // initialize ncurses window and draw the maze
  while ( c != 'q' ) {   // as long as the user doesn't press q ..
    clearScreen();
    draw(x, y, '@', 2);   // draw our player and maze, check for collisions
    c = getch();          // capture the user's pressed key
    switch (c) {
      case 'w': y--; break;  // go up
      case 's': y++; break;  // go down
      case 'a': x--; break;  // go left
      case 'd': x++; break;  // go right
    }
  }
  endwin();           // ncurses function: close the ncurses window
  return 0;
}
```

5.3. Arrays: Strings (Arrays of char)

- Strings are sequences of symbols, for example to store textual data
- In C++, there is no built-in (primitive) string type. Sequences of characters can easily be implemented as an **array** of **`char`** variables, which *always* end with a zero (a character that has the value **`0`**, or also: **`'\0'`**, but NOT **`'0'`**):

```
char myName[10] = {'S', 'l', 'i', 'm', 'S', 'h', 'a', 'd', 'y', 0};
std::cout << yourName << std::endl;  // returns contents of yourName
```



5.3. Arrays: Strings (Arrays of char)

- Later, we will see that :

```
char yourName[] = "Marshall Bruce Mathers III";  // works, too, and
                                                 //   ends with a 0
```

- We have already used constant strings when writing output for the terminal:

```
#include <iostream>
std::cout << "This is a string!" << std::endl;
```

- The ending zero (which also is present in the constant strings such as these two above) makes sure that we never go beyond the end of the string
- As such, the empty string `""` contains still one character (with value `0`, or also: `'\0'`, but NOT `'0'`)

5.3. Arrays: Strings (Arrays of char)

- With arrays of characters, you can manage any string already, but you will see that strings are not as easy to deal with as the basic types (`int`, `float`, `double`, `bool`, `char`). For example concatenating two strings is lots of work:

```
/** Write a program that concatenates two strings, s1 and s2, no matter
    what size they have */
#include <iostream>  // use std::cout, std::cin, and std::endl
int main() {
  char s1[] = "Apples and ", s2[] = "oranges";
  // create a new string s, which contains s1 and s2 below:

  std::cout << "Concatenated string: " << s << std::endl;
  return 0;
}
```

5.4. Arrays as function parameters

- In C++, array parameters are passed ***by reference***

```
void swap( int a[10], int i, int j) {  // this swap function works!
  int temp = a[i];  // after this function ends, the original array a
  a[i] = a[j];      // will have swapped the values in its elements i
  a[j] = temp;      // and j. Variables i, j, and temp were created
}                   // at function start and are removed from memory
```

- The function above thus uses the actual array parameter, not a copy
- With ***call-by-reference***, variables given as actual parameters may be changed by the function
- In a function declaration, arrays can be of unspecified length:

```
void swap( int a[], int i, int j);  // Note we'll have to check for a's size
```

5.5. Reading char arrays from the terminal

- When trying our this approach:

```
char buffer[80];
std::cin >> buffer;
```

  you will see this has a few flaws: **`cin`** stops reading beyond the first whitespace character (so we cannot input sentences), and we might have a buffer overrun when we enter more than 80 characters

- The correct approach is to use:

```
char buffer[80];
std::cin.get( buffer, 80 );  // Reads at most 79 characters, 0 is last element
```

- In the above, **`get()`** seems to be a function, but: What exactly is **`cin`**?

5.6. Lambda Expressions and Range-based Loops (since C++11)

- Lambda expressions construct a **closure**: an unnamed function object that is capable of capturing variables in scope
- These are typically used for short code snippets that are not reused and therefore do not specifically require a name:

```
auto x = [](char symbol) { std::cout << symbol << ' '; };
```

```
auto x = [](double d, int t) -> double { return (d<t)?0:d; };
```

capture clause (see later) parameters return type function body

5.6. Lambda Expressions and Range-based Loops (since C++11)

- Lambdas are the simplest way of passing functions as arguments, two other methods are (1) passing functions as pointers and (2) using the std::function<> template class → see [more in-depth information] or later in this course

5.6. Lambda Expressions and Range-based Loops (since C++11)

- The foreach loop or range-based for loop eases iterating over data
- It leaves out the iterator, initialization and stopping conditions:

```
#include <iostream>  // output to the console
int main() {
  int array[]= { 8, 2, 7, 2, 8, 7, 9, 1};
  for( auto value : array ) {  // foreach loop over array
    std::cout << value << ' ';
  }
  std::cout << std::endl;
  return 0;
}
```

5.6. Lambda Expressions and Range-based Loops (since C++11)

- **`std::for_each`** loops are similar to range-based for loops, and provided in **`<iostream>`**
- They apply a *function* to each of the elements in the range [first,last):

```
#include <iostream>  // output to the console, for_each
int main() {
  char array[] = {'H', 'e', 'l', 'l', 'o', '?'};
  std::for_each(std::begin(array), std::end(array),
                [](char sym) { std::cout << sym << 'l'; });
  std::cout << std::endl;
  return 0;
}
```