Advanced Programming in C++

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

```cpp
#include <iostream>
#include <vector>

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

Contents

Contents

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

introduction to programming in C++

advanced programming in C++

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 courses from colleagues Roland Wismueller (Uni Siegen), Hannah Bast (Uni Freiburg), Federico Busato (NVIDIA, Modern C++ Programming)

See the description of this course here

Enroll for both lecture *and* exercises, as well as the course work (4INFMA307-S):

- Set of programming assignments during term: programming on paper
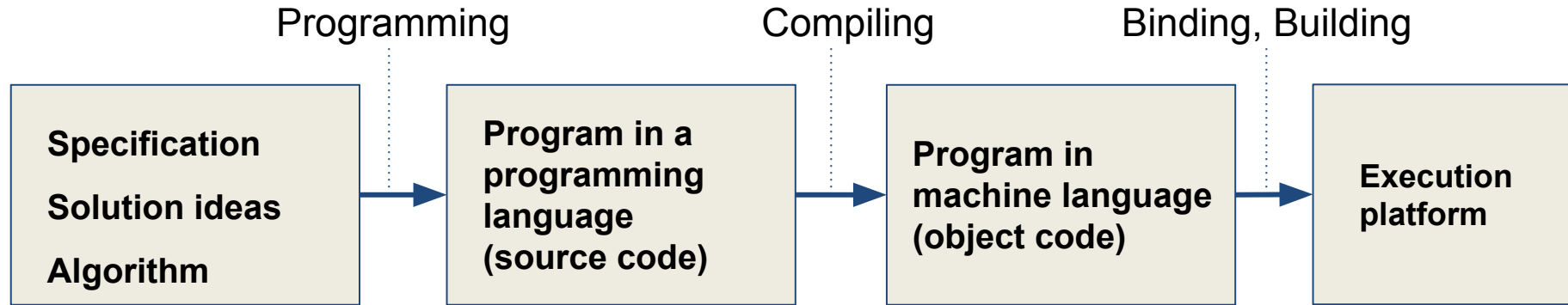- These need to be delivered in-class => your presence required
- You need to pass this course work to be able to enroll for the exam

A 1-hour written exam (enroll more than 2 weeks before: 4INFMA307-P):

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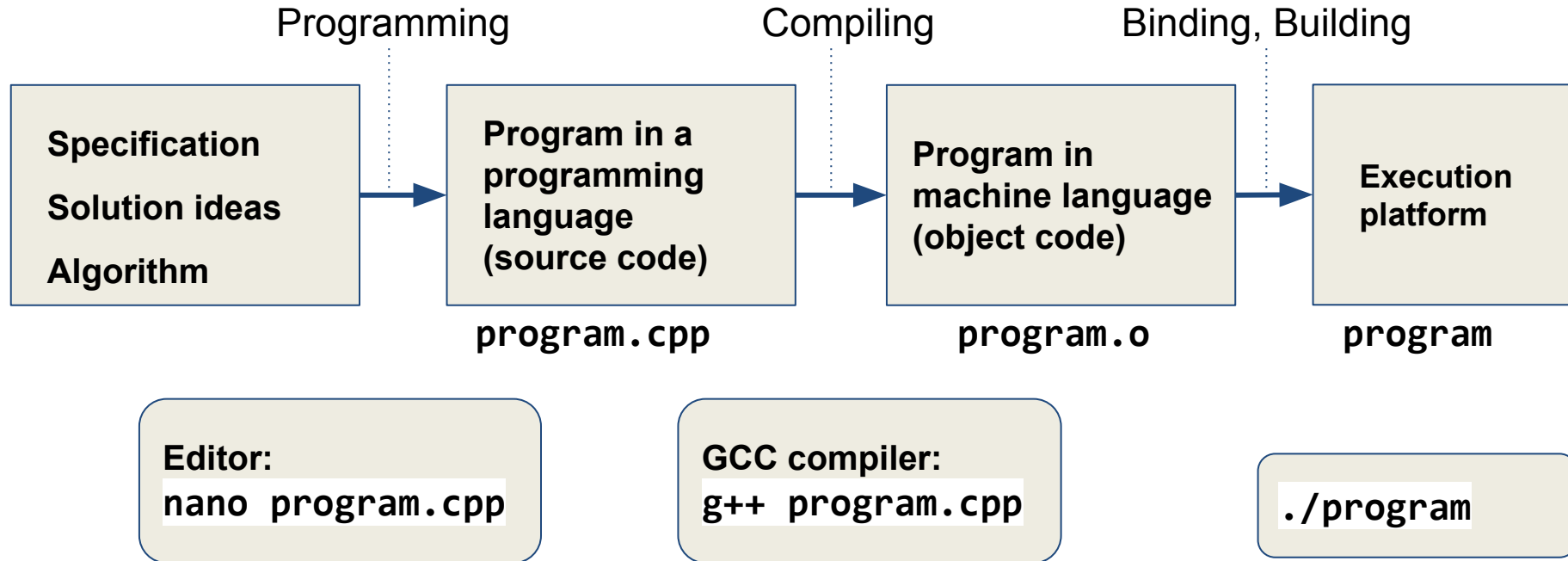- Structure: Programming tasks *very similar* to the exercises

UNIVERSITÄT SIEGEN

Ubiquitous Computing

Steps in creating a program:

| Programming | | Compiling | | Binding, Building |

```
┌─────────────────┐     ┌─────────────────┐     ┌─────────────────┐     ┌─────────────────┐
│                 │     │                 │     │                 │     │                 │
│ Specification   │ ──► │ Program in a    │ ──► │ Program in      │ ──► │ Execution       │
│                 │     │ programming     │     │ machine language│     │ platform        │
│ Solution ideas  │     │ language        │     │ (object code)   │     │                 │
│                 │     │ (source code)   │     │                 │     │                 │
│ Algorithm       │     │                 │     │                 │     │                 │
└─────────────────┘     └─────────────────┘     └─────────────────┘     └─────────────────┘
```

UNIVERSITÄT SIEGEN

Ubiquitous Computing

Steps in creating a program:

Programming   Compiling   Binding, Building

Specification **Solution ideas** **Algorithm**	→	**Program in a programming language (source code)**	→	**Program in machine language (object code)**	→	**Execution platform**

                `program.cpp`   `program.o`   `program`

Editor:
`nano program.cpp`

GCC compiler:
`g++ program.cpp`

`./program`

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

1.2. 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

1.2. 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

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

```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 of " << n << " people have a same birthday is " << b.prob(n);
  return 0;
}
```

BDay.cpp

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**

1.3. 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

```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 \n";
  return 0;
}
```

BDaySimple.cpp

1.4. Including libraries:

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

    g++ BDayCentre.cpp -l ncurses

```cpp
/* The Birthday Paradox -- in the *middle*  */        BDayCentre.cpp
#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;
}
```

1.5. 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):

```cpp
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 }
```

2.1. The basic components of a program:

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

```cpp
/* 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 << '\n';
  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

2.1.4. 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

2.1.8. 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`

2.1. Example:

```cpp
/* An interactive example */
#include <iostream>
int main() {
  char name[80];   // symbols array for the user's name
  std::cout << "Hi there, what's your name?\n";
  std::cin >> name;  // read the name from terminal
  std::cout << "Welcome " << name;
  if (name[0] == 'K') {
    std::cout << ", I like your name!"
  }
  std::cout << '\n';
  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	see **ASCII** table	
signed char	1	-128 -- 127	`int8_t`
unsigned char	1	0 -- 255	`uint8_t`
short	2	-2^{15} -- 2^{15}-1	`int16_6`
unsigned short	2	0 -- 2^{16}-1	`uint16_t`
int	4	-2^{31} -- 2^{31}-1	`int32_t`
unsigned int	4	0 -- 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} -- 2^{63}-1	`int64_t`
long long unsigned int	8	0 -- 2^{64}-1	`uint_64_t`

2.3. Data types: Floating-Point

| Native type	Bytes	Range	Fixed width types C++23 <stdfloat>
(bfloat16)	2	$\pm1.18 \times 10^{-38}$ -- $\pm3.4 \times 10^{38}$	`std::bfloat16_t`
(float16)	2	0.00006 -- 65536	`std::float16_t`
float	4	$\pm1.18 \times 10^{-38}$ -- $\pm3.4 \times 10^{38}$	`std::float32_t`
double	8	$\pm2.23 \times 10^{-308}$ -- $\pm1.8 \times 10^{308}$	`std::float64_t`

○ See also **IEEE754** for more information

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
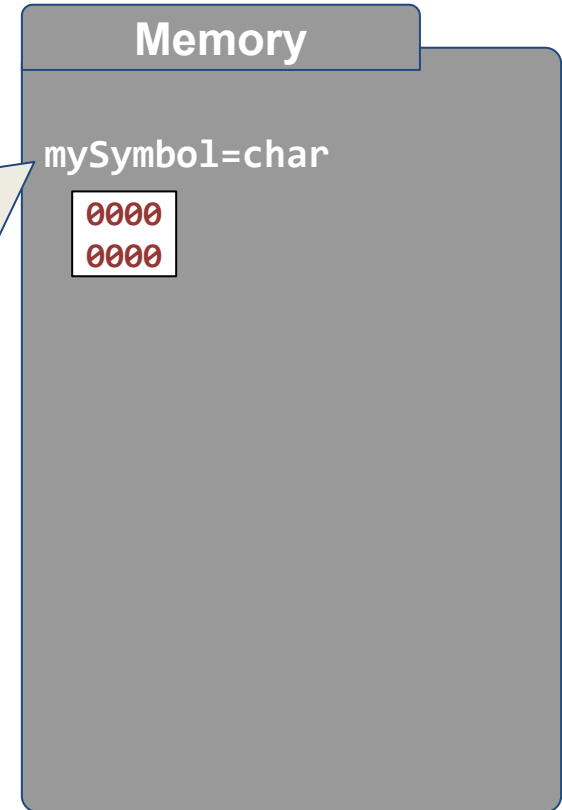 - for variables, this is explicit in the declaration

2.3. Data types: A (simplified) Memory View
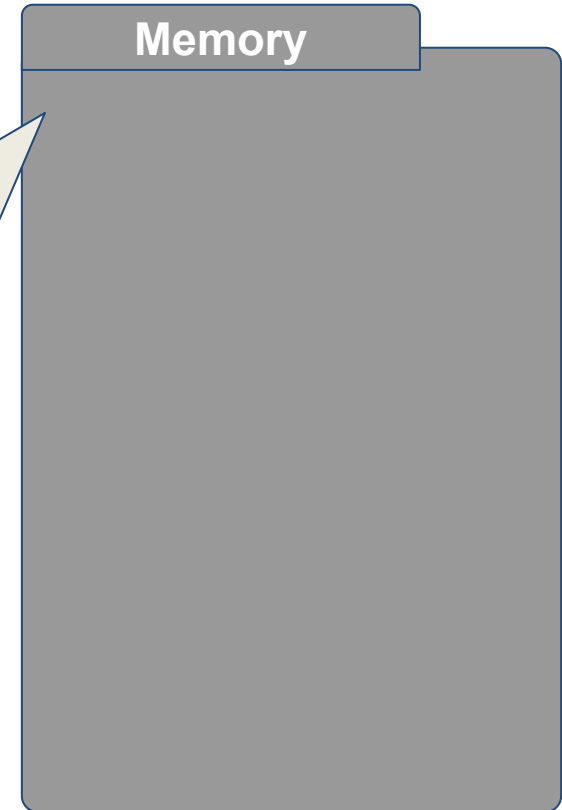
Memory

```
/* 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

```c
/* 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;
}
```
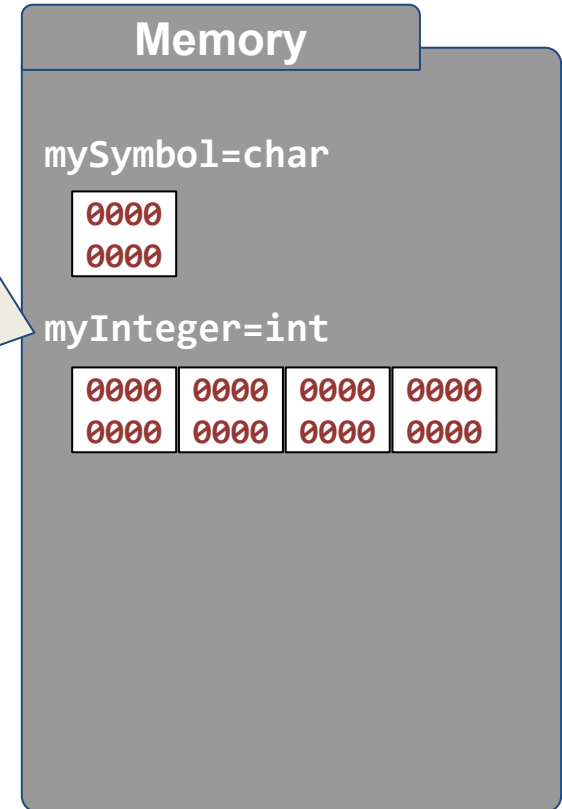
Memory

mySymbol=char

0000
0000

2.3. Data types: A (simplified) Memory View

```c
/* 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;
}
```

Memory

mySymbol=char

0000
0000

myInteger=int

0000	0000	0000	0000
0000	0000	0000	0000

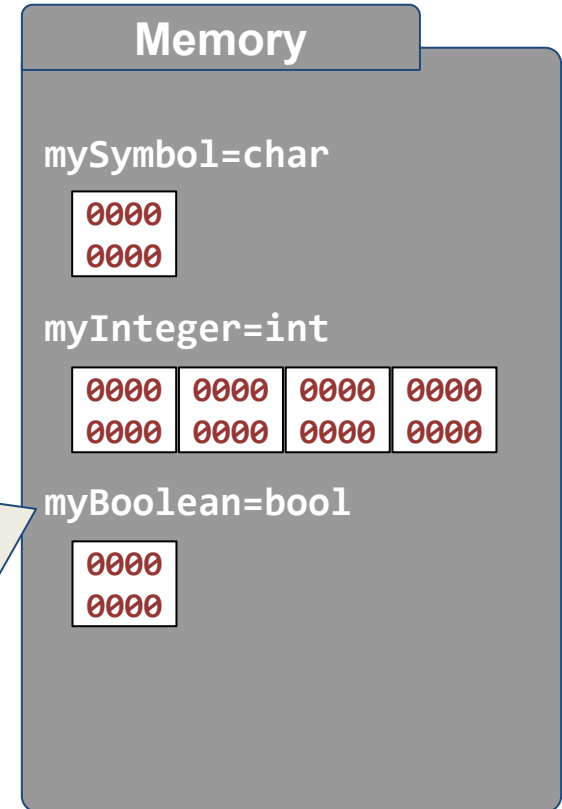## 2.3. Data types: A (simplified) Memory View

```
/* 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;
}
```

Memory

mySymbol=char

| 0000 |
| 0000 |

myInteger=int

| 0000 | 0000 | 0000 | 0000 |
| 0000 | 0000 | 0000 | 0000 |

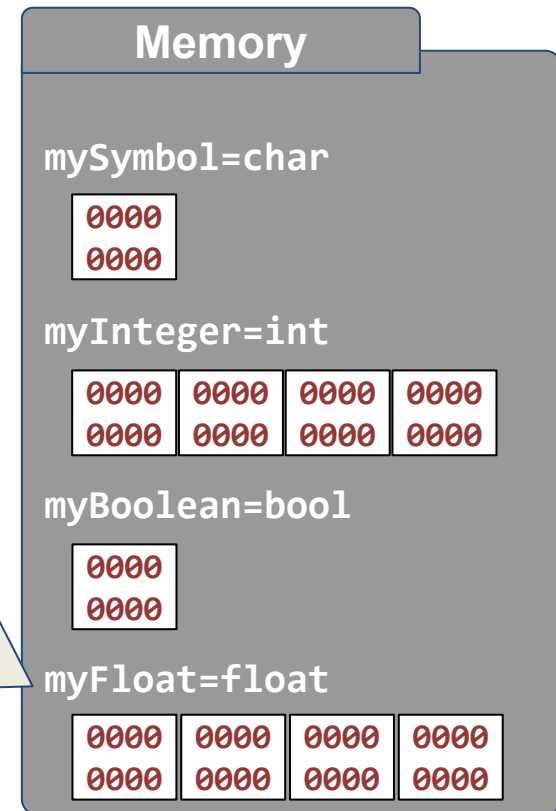myBoolean=bool

| 0000 |
| 0000 |

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;
}
```
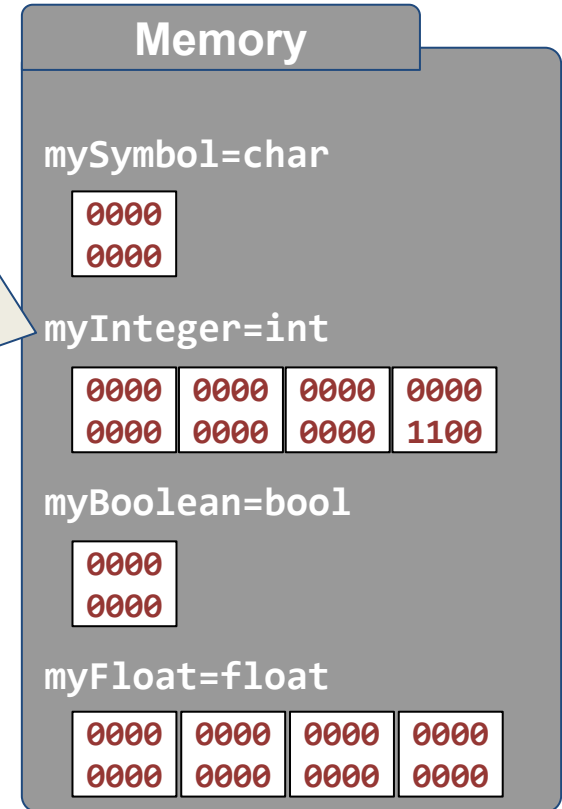
Memory

mySymbol=char

0000
0000

myInteger=int

0000	0000	0000	0000
0000	0000	0000	0000

myBoolean=bool

0000
0000

myFloat=float

0000	0000	0000	0000
0000	0000	0000	0000

2.3. Data types: A (simplified) Memory View

```c
/* 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;
}
```

Memory

mySymbol=char

0000
0000

myInteger=int

0000	0000	0000	0000
0000	0000	0000	1100

myBoolean=bool

0000
0000

myFloat=float

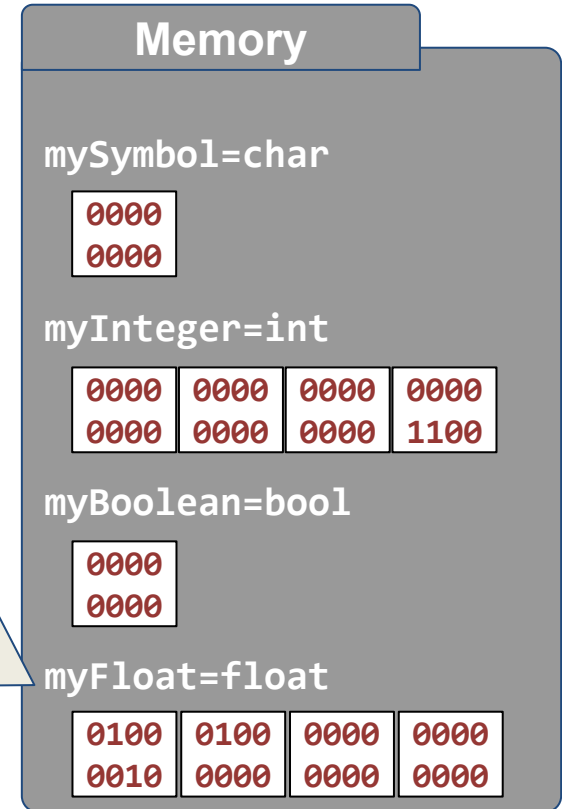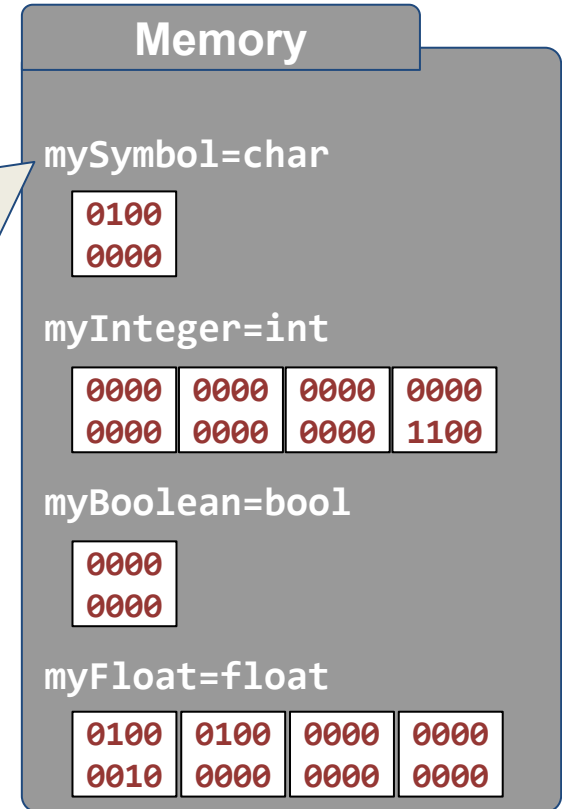| 0000	0000	0000	0000
0000	0000	0000	0000

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;
}
```

Memory

mySymbol=char

0000
0000

myInteger=int

0000	0000	0000	0000
0000	0000	0000	1100

myBoolean=bool

0000
0000

myFloat=float

0100	0100	0000	0000
0010	0000	0000	0000

2.3. Data types: A (simplified) Memory View

```c
/* 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;
}
```
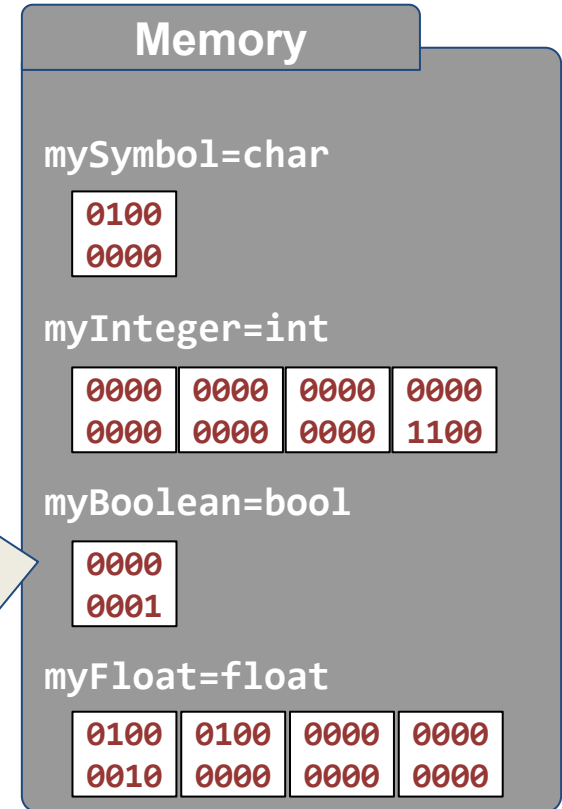
Memory

mySymbol=char

0100
0000

myInteger=int

0000	0000	0000	0000
0000	0000	0000	1100

myBoolean=bool

0000
0000

myFloat=float

0100	0100	0000	0000
0010	0000	0000	0000

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;
}
```

Memory

mySymbol=char

0100
0000

myInteger=int

0000	0000	0000	0000
0000	0000	0000	1100

myBoolean=bool

0000
0001

myFloat=float

0100	0100	0000	0000
0010	0000	0000	0000

2.3. Data types: Constants

Constants for these type are visible from how they are written, mostly using a type-dependant *suffix*:

- `int` whole numbers without suffix, e.g.: **729**, **-3628**, **0**, **-12632832**
- `unsigned int` **u** or **U**, e.g.: **56u**, **7126u**, **0U**
- `long int` **l** or **L**, e.g.: **523374631**, **-363433428L**
- `long unsigned` **ul** or **UL**, e.g.: **39ul**, **3637428UL**
- `long long int` **ll** or **LL**, e.g.: **52ll**, **1634428LL**
- `long long unsigned` **ull** or **ULL**, e.g.: **7ull**, **8428ULL**
- `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: Constants

Constants for C++23 float types use a type-dependant **suffix**:

- `std::bfloat16_t`   **bf16** or **BF16**,   `3.21bf16`, `-12.345BF16`
- `std::float16_t`   **f16** or **F16**,   `3.21f16`, `-12.345F16`
- `std::float32_t`   **f32** or **F32**,   `3.21f32`, `-12.345F32`
- `std::float64_t`   **f64** or **F64**,   `3.21f64`, `-12.345F64`
- `std::float128_t`   **f128** or **F128**,   `3.21f128`, `-12.345F128`

Constants for number types can use a **prefix** for different representations:
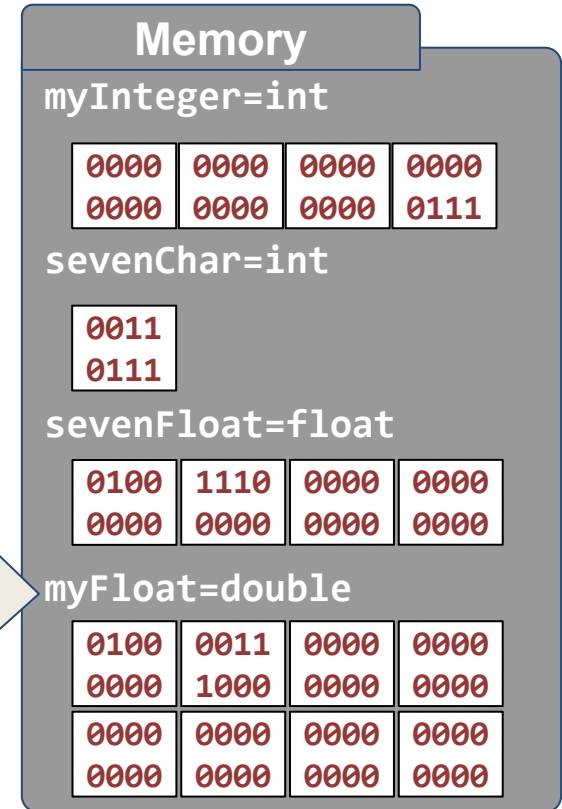
- Binary (since C++14): **0b**,   `0b10101010`
- Octal: **0**,   `0308`
- Hexadecimal: **0x** or **0X**,   `0xFA77DD`, `0X1A7F`

Since C++14, digit separators `'` can be used: `80'000'000` (=80000000)

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;
}
```

Memory

myInteger=int

| 0000 | 0000 | 0000 | 0000 |
| 0000 | 0000 | 0000 | 0111 |

sevenChar=int

| 0011 |
| 0111 |

sevenFloat=float

| 0100 | 1110 | 0000 | 0000 |
| 0000 | 0000 | 0000 | 0000 |

myFloat=double

0100	0011	0000	0000
0000	1000	0000	0000
0000	0000	0000	0000
0000	0000	0000	0000

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

A first console output example: Initialize a variable to the symbol **a**, print its value in the console, then change its value to a **q** , then print it again.

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

2.3. Data types and variables: `auto`

Since C++11, the `auto` keyword can be used to let the compiler *deduce* the type from its initialization:

```
auto mySymbol = '&';            // a char
auto myNumber = 2 + 2;          // sum of two ints -> int
auto areWeDone = 7 < 3;         // false -> bool
auto heightInMeters = 1.85f;    // float
auto highPrecision = 1.2345;    // double
```

The auto keyword can make code more maintainable and hide complexity:

```
for ( auto i = start; i < end; i++ ) ...   // i has type of start
```

⚠ Excessive use of type hiding typically makes code less readable.

2.3. Data types and variables: `decltype`

From C++11 onwards, the `decltype` keyword can be used to inspect the declared type of an entity or expression:

```cpp
auto myNumber = 2 + 2;                    // sum of two ints -> int
decltype(myNumber) otherNumber = myNumber + 3;  // -> int
decltype('?') mySymbol = '?';
```

STL-based (see later) ways to query type:
- The `typeid` operator can be used to identify the type:
```cpp
typeid(myNumber).name() == 'i';   // returns true
```

- `std::is_same` can be used to check whether two type are the same:
```cpp
std::is_same<int, std::int32_t>::value;  // returns mostly true
```

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
- Sticking to **explicit conversion**, using the type in braces, is clearer:

```cpp
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 '#'

```cpp
#include <iostream>  // to allow use of std::cout
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 << '\n';   // print ASCII code for '?'
  std::cout << (int)ampersand << '\n';  // print ASCII code for '&'
  std::cout << (int)hash << '\n';       // print ASCII code for '#'
  return 0;
}
```

3.1. Statements and assignments

3.2. Priority

3.3. Blocks of statements

3.4. `if` and `switch`

3.5. Repetitions / loops:

    3.5.1. the `while` loop

    3.5.2. the `do while` loop

    3.5.3. the `for` loop

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)
```

3.1. 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   unary and binary operators

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:

Important 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:

Relational and logical operators (try out yourself):

```cpp
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: 🌶️🌶️):

```cpp
/**
 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 << '\n';
  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 = a / ((b * c) % d)
a += b = c + d;     // a = 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 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:

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

```cpp
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 a nested **(? :)** 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:

```c
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:

```c
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   do we   in- or decrease
loop variable   loop again?   the loop variable

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

statement, or block of statements to repeat
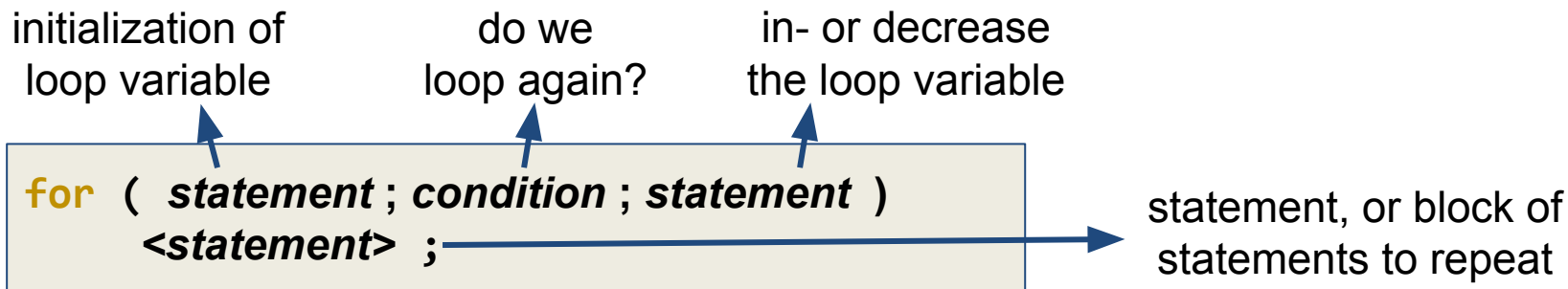
3.5. Loops: **break** and **continue**

- **break** terminates the 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: 🌶️)

```cpp
/**
  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: 🌶️🌶️)

```cpp
/**
  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: 🌶️🌶️)

```cpp
/**
  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: 🌶️🌶️🌶️)

```cpp
/**
 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               XX X
   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;
}
```

4.1. 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. Variadic arguments

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:

```c
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:

```cpp
#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 << '\n';
}
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:

```cpp
int a();  // declaration of function a()
int b();  // declaration of function b()
int a() {  // implementation of function a():
  std::cout << "Yes" << '\n';
  return b();
}
int b() {  // implementation of function b():
  std::cout << "No"<< '\n';
  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 << '\n';    // 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;
}
```
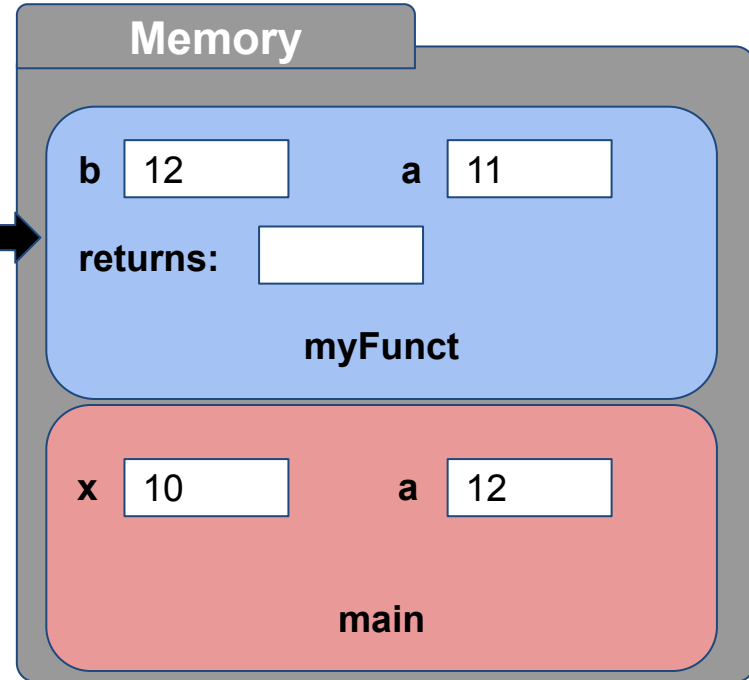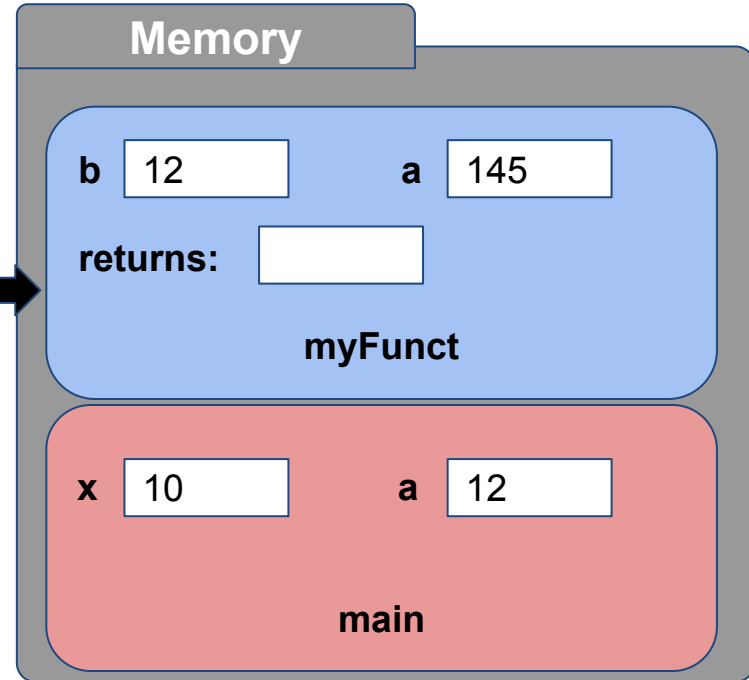
Memory

x `10`   a `12`

main

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;
}
```

Memory

b 12   a 11

returns: [   ]
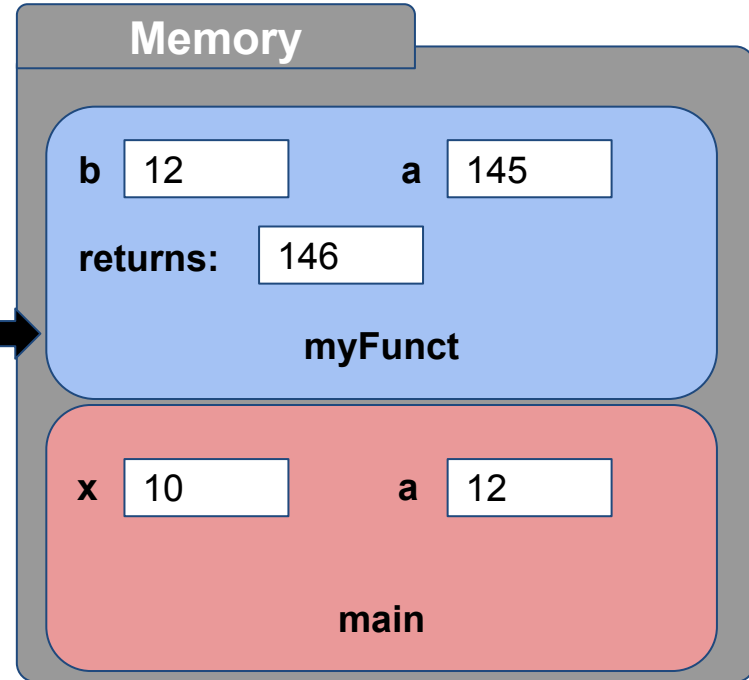
myFunct

x 10   a 12

main

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;
}
```

Memory

b `12`   a `11`

returns: `_____`

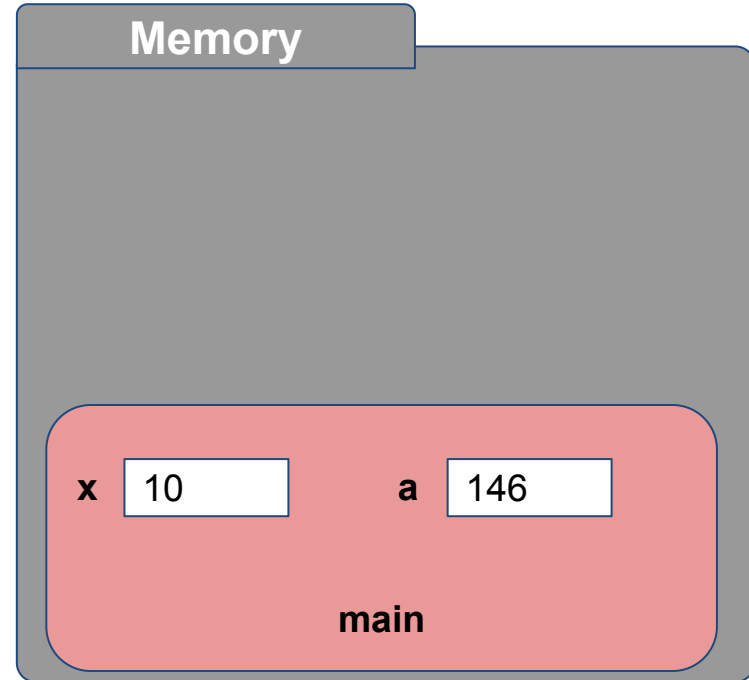**myFunct**

x `10`   a `12`

main

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;
}
```

Memory

b 12   **a** 145

returns:

myFunct

x 10   **a** 12

main

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;
}
```

Memory

b 12   a 145

returns: 146

myFunct

x 10   a 12

main

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;
}
```

Memory

x 10   a 146

main

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

```c
/* 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

```cpp
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

```cpp
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;
}
```

4.1. Functions and their parameters: Using Functions

Bluetooth.cpp (difficulty level: 🌶🌶🌶🌶): Draw a bluetooth icon of a particular odd width, in ncurses. Draw spaces in white on a blue background. Use `int width` as a parameter and only draw the icon when `width` is odd.

```cpp
#include <ncurses.h>  // functions to draw colored text
// --- implement the bluetooth function here ---
int main() {
  initscr(); curs_set(0);  // initialize window, hide cursor
  noecho();  // don't show keys pressed in terminal
  start_color();  // use color
  init_pair(1, COLOR_BLACK, COLOR_BLUE);
  init_pair(2, COLOR_BLACK, COLOR_WHITE);
  bluetooth(9);  // draw a bluetooth icon of width 9
  auto c = ' '; while ( c != 'q' ) c = getch();  // wait for 'q'
  endwin();  // ncurses function: close the ncurses window
}
```

for width 5:
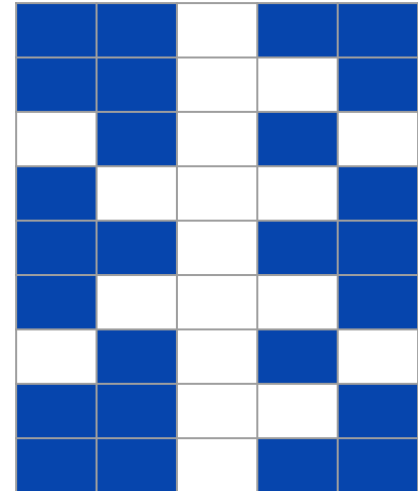
4.2. Recursive Functions

- 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);
```

Memory
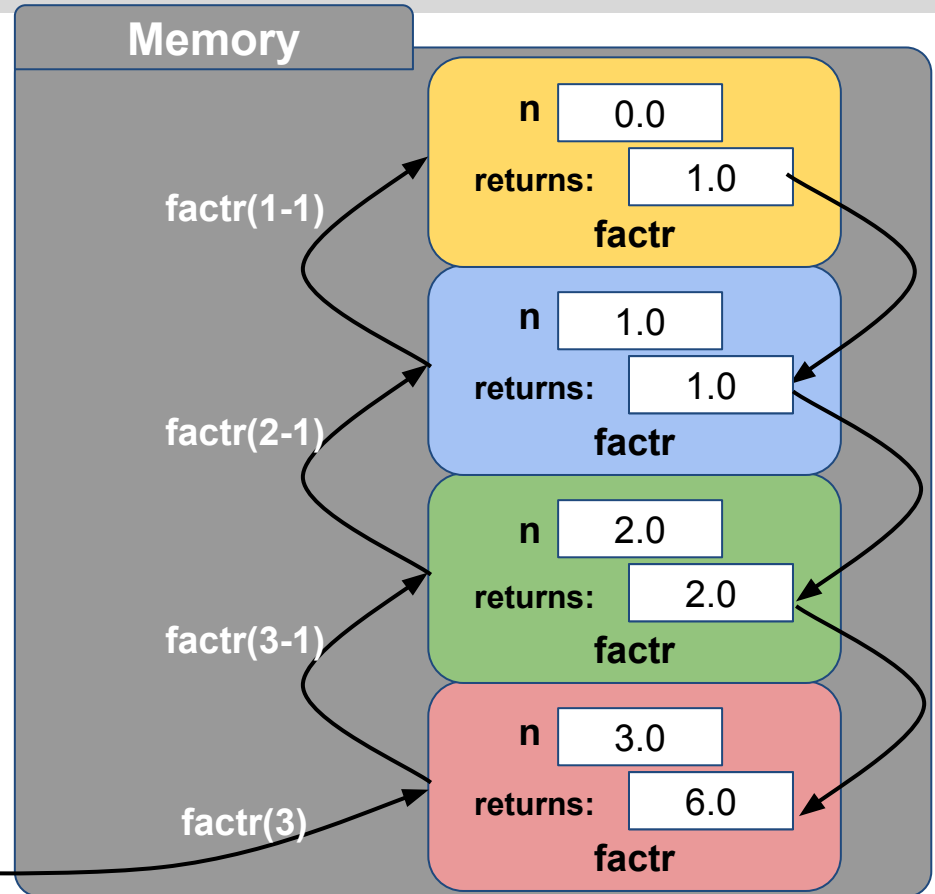
4.3. Call by Value

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:

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

```cpp
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:

```cpp
int a = 10;
y = maximum(a, a);       // the value 10 is copied to both parameters
```
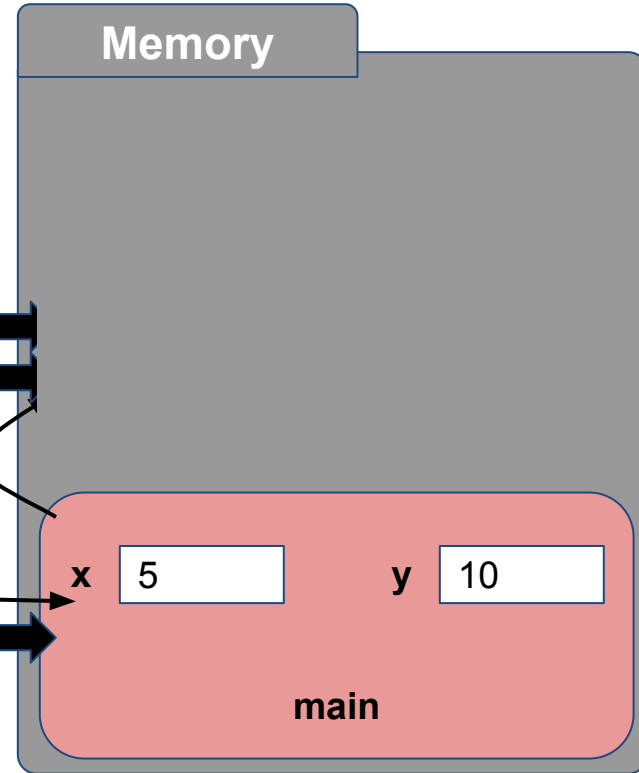
4.3. Call by Value

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 << '\n';
   return 0;
}
```
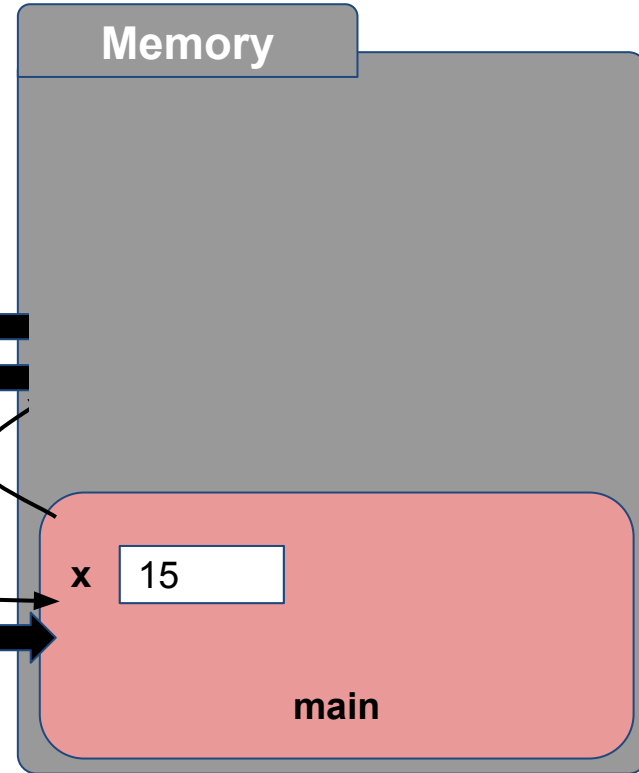
Memory

x 5   y 10

main

4.3. Call by Value

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

You can use the function's return value:

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

Memory

x 15

main

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)

```cpp
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:

```cpp
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

```cpp
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

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

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

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

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

```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

```
/* 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);
```

drawMaze.h

4.6. Header files and Modules

```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));
}
```
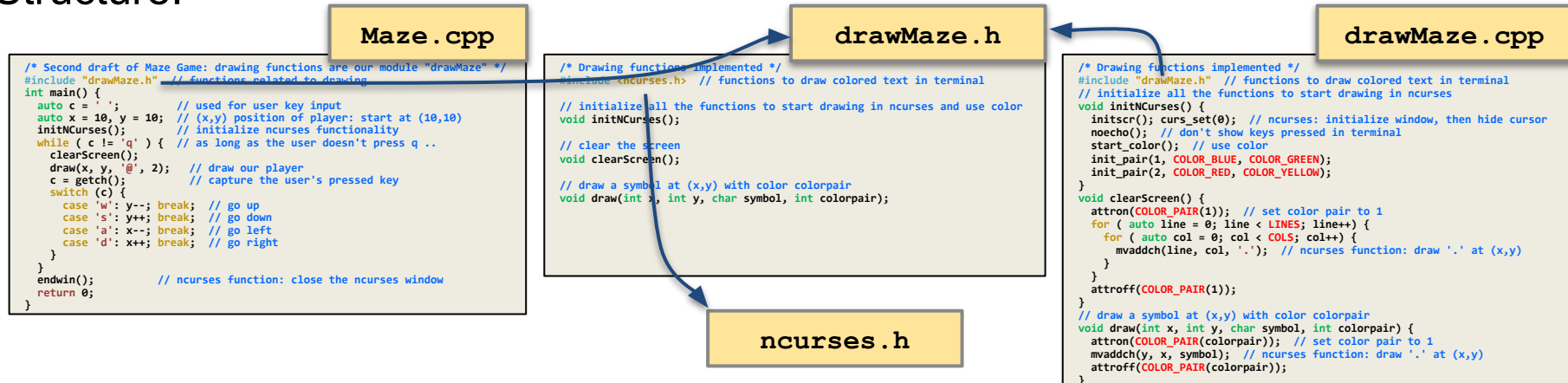
drawMaze.cpp

4.6. Header files and Modules

drawMaze.cpp

```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:

Maze.cpp

```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;
}
```

drawMaze.h

```cpp
/* Drawing functions implemented */
#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);
```

ncurses.h

drawMaze.cpp

```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));
}
// 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));
}
```

4.6. Header files and Modules
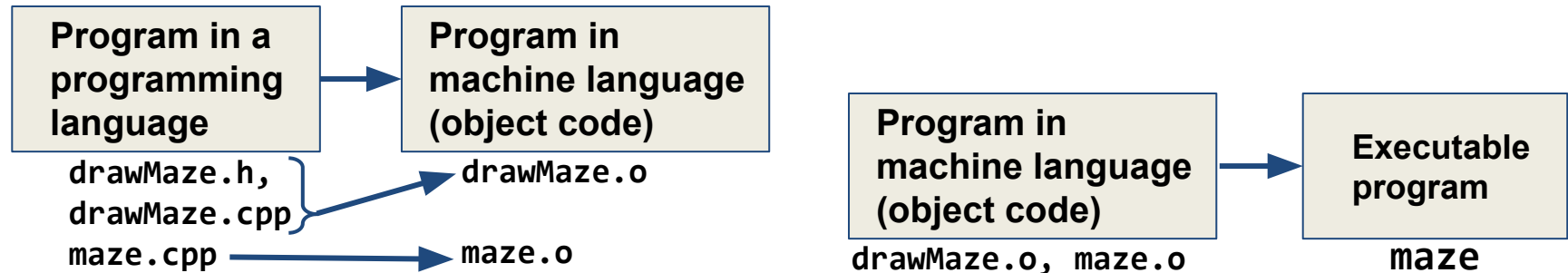
Maze Game v.2.0: How to compile the program?

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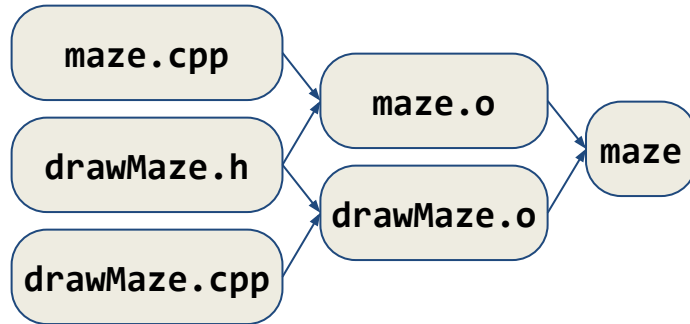
  `g++ maze.o drawMaze.o -o maze -l ncurses`

Program in a programming language	Program in machine language (object code)

`drawMaze.h, drawMaze.cpp` → `drawMaze.o`

`maze.cpp` → `maze.o`

Program in machine language (object code)	Executable program

`drawMaze.o, maze.o` → `maze`

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++11s`

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**


```
# Rule to make our program when     Makefile
# '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
```
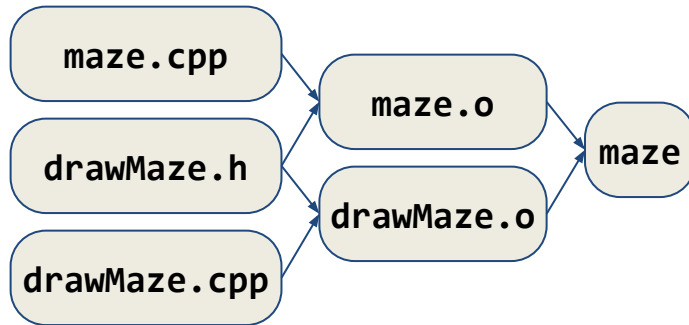
- After each rule, we need to type a **tab** before each g++ command in **Makefile**

4.7. Variadic arguments

- Functions can take a variable number of parameters, using an ellipsis (**...**) as the last argument/parameter (example: see **std::printf**)
- Within the body of the variadic function, the values of these arguments can be accessed, using these function macros and type from the **<cstdarg>** library:
  - **va_start**: enables access to variadic function arguments
  - **va_arg**: accesses the next variadic function argument
  - **va_copy** (since C++11): makes a copy of the variadic arguments
  - **va_end**: ends traversing through the variadic arguments
  - **va_list**: holds the information needed by the above function macros

4.7. Variadic arguments

● Example: (traversing the `format` string by pointer -- see next chapters)

```cpp
void myPrint(const char * format, ...) {
  va_list args;
  va_start(args, format);
  while (*format != '\0') {
    int i = va_arg(args, int);
    if (*format == 'd') {
      std::cout << 'i' << i << ' ';
    } else if (*format == 'c') {
      std::cout << 'c' << (char)i << ' ';
    }
    ++format;
  }
  va_end(args);
}
```

```cpp
#include <cstdarg>
#include <iostream>

int main() {
  myPrint("dcd", 3, 'a', 14);
  myPrint("cc", 'c', 'd');
  std::cout << '\n';
}
```

Summary

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

- A function returns at most one value and thus must have a return type (so `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 typed variables (`int`, `float`, `double`, `bool`, `char`, etc.)
- 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;
  }
}
```

5.1. 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:

Memory						
element 1 *4 bytes*	*element 2* *4 bytes*	*element 3* *4 bytes*	*element 4* *4 bytes*	*element 5* *4 bytes*	*element 6* *4 bytes*	*element 7* *4 bytes*
`myArray[0]`	`myArray[1]`	`myArray[2]`	`myArray[3]`	`myArray[4]`	`myArray[5]`	`myArray[6]`
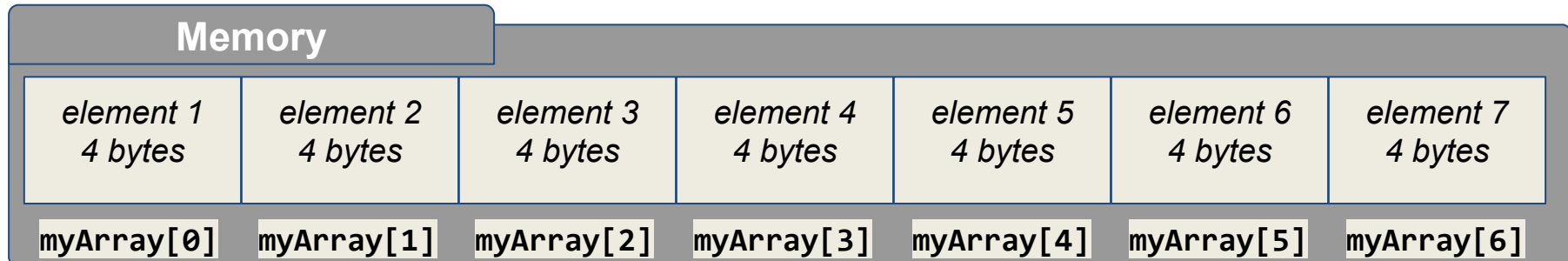
5.1. Arrays: Initialization, sizeof

- 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]);  // 32/8
```

- 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;
```
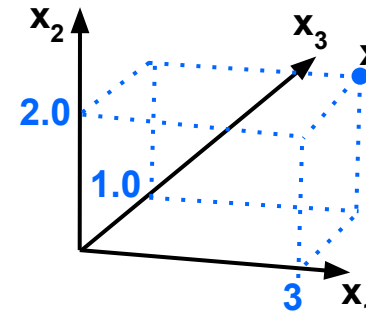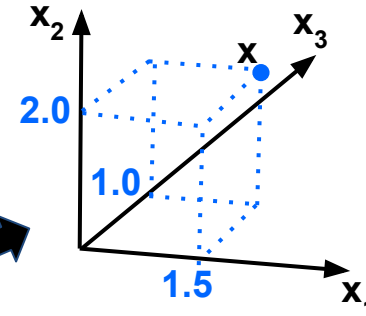
5.1. Arrays: Writing beyond the array boundary

- 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:

```cpp
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:

Memory				
9	8	7	6	5
myArray[0]	myArray[1]	myArray[2]	myArray[3]	myInteger
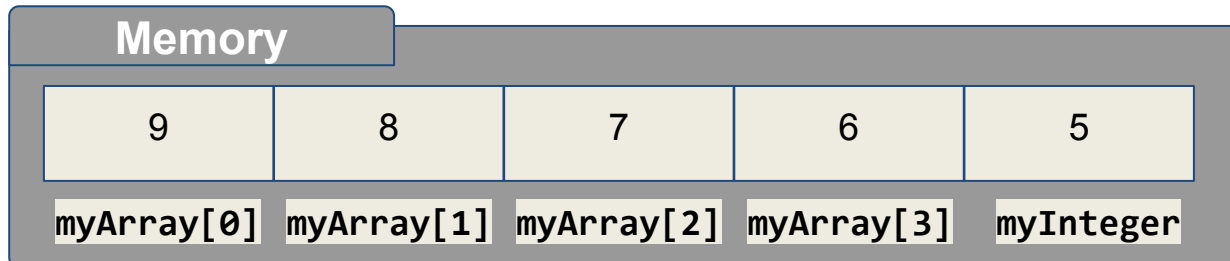
5.1. Arrays

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 and a loop iteration variable.
*/

int main() {
  bool myArray[50];



  return 0;
}
```

5.1. Arrays

Example 02 (difficulty level: 🌶️🌶️)

```cpp
/**
  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] << '\n';
  }
  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 `clearScreen` function to build up a maze, for example:

```c
int 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

```c
/* 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
}
```

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 << myName << '\n';  // returns contents of myName
```

Memory									
'S'	'l'	'i'	'm'	'S'	'h'	'a'	'd'	'y'	'\0'
myName[0]	myName[1]	myName[2]	myName[3]	myName[4]	myName[5]	myName[6]	myName[7]	myName[8]	myName[9]
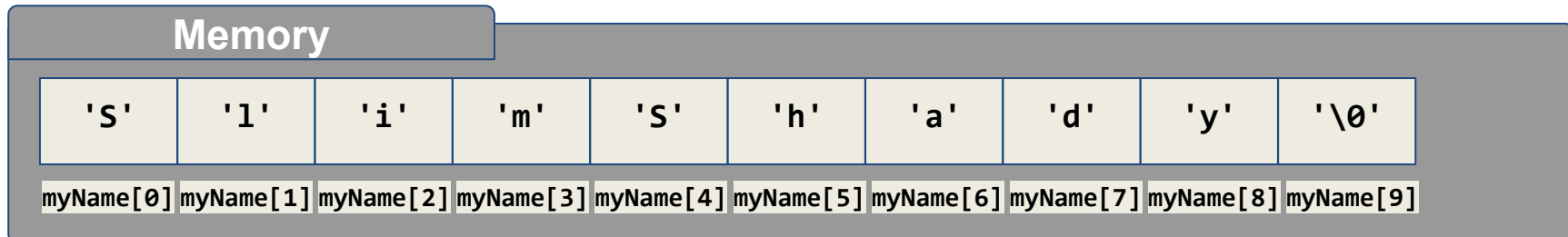
5.3. Arrays: Strings (Arrays of char)

- Constant C strings can be used for initializing:
```cpp
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:
```cpp
#include <iostream>
std::cout << "This is a string!\n";
```

- 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:

```cpp
/** 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 << '\n';
}
```

5.4. Arrays as function parameters

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

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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 (since C++11)

- Lambda expressions construct a **closure**: an unnamed function object that is capable of capturing variables in scope
- They are often used as callbacks (functions as arguments), for example when iterating over containers such as arrays (see also STL later)
- These are typically used for short code snippets that are not reused (they are **inline**) and do not specifically require a name:

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

```cpp
auto x = [](double d, int t) -> double { return (d<t)?0.0:d; };
```

capture clause (see next slide)   parameters   return type   function body

5.6. Lambda Expressions (since C++11)

- We can capture external variables from the enclosing scope in three ways using the **capture clause**:
  - **[&]**: capture all external variables by reference
  - **[=]**: capture all external variables by value
  - **[a, &b]**: capture variable **a** by value, and variable **b** by reference

```cpp
int a = 7, b = 14;
auto swap = [&a, &b]() -> { int t = a; a = b; b = t; };
```

- 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 (since C++11)

- Example:

```cpp
#include <iostream>  // output to the console
int main() {
  int a = 7, b = 14;
  // passing a and b by reference:
  auto swap = [&a, &b]() { int t = a; a = b; b = t; };
  // Note that this won't work:
  // auto swap = [a, b]() { int t = a; a = b; b = t; };
  swap();
  std::cout << "a = " << a << '\n';
  std::cout << "b = " << b << '\n';
}
```

5.7. Range-based Loops (since C++11)

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

```cpp
#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 << '\n';
}
```

5.7. Range-based Loops (since C++11)

- for multi-dimensional arrays, foreach loops look like this (using *references* → see later in chapter 7):

```cpp
#include <iostream>  // output to the console
int main() {
  int array[][]= { {8, 2, 7}, {2, 8, 7}, {9, 1, 0} };
  for ( auto & row : array ) {         // loop over 2d array rows
    for ( auto & element : row ) {  // loop over row's elements
      std::cout << element << ' ';
    }
  }
  std::cout << '\n';
}
```

5.7. Range-based Loops (since C++11)

- Example 04 (difficulty level: 🌶️🌶️)

```cpp
#include <iostream>  // output to the console with std::cout
int main() {
  int myArray[7][7];

  // use foreach loops to initialize myArray, so that each
  // element's value is one higher than the previous:


  // print myArray using foreach loops, one row per line,
  // zero-pad the elements if they are smaller than 10:


}
```

5.7. 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):

```cpp
#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 << ' '; });
  std::cout << '\n';
}
```

6.1. Classes

6.2. Constructors and Destructors

6.3. `this` and initializing `const` attributes

6.4. `static` members

6.5. The `string` Class

6.6. The `ifstream` and `ofstream` Classes

6.7. The `tuple` Class

6.1. Classes and Objects

● A **class** essentially defines a new type, containing:

   ○ a collection of variables (data members, attributes)

   ○ a set of related operations (member functions, methods)

● An **object** is an instance (an entity) of a class

   ○ it is called object, since it usually models a real-world object

   ○ a class then can be viewed as a model or a blueprint for a certain class of objects

6.1. Classes and Objects

Encapsulation: The bundling together of all information, capabilities, and responsibilities (data and functions) into one single object:

```cpp
// create a new employee:
Employee user("Carnegie", 62413,
              12, 4729.12);

// move the user to department 17:
user.movetoDept(17);
// change the user's name:
user.changeName("Carnegie-Mellon");
```
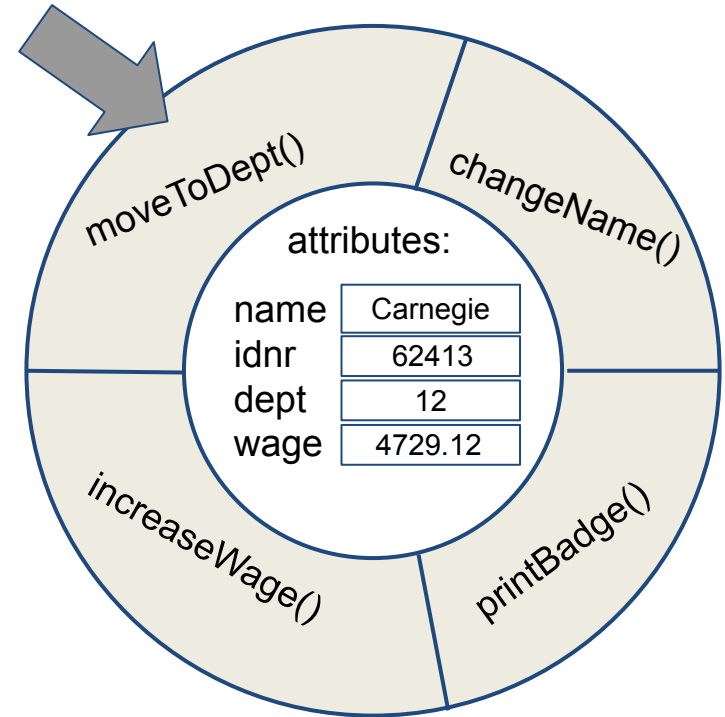
6.1. Classes and Objects

Encapsulation has two properties:

1) Data protection: Attributes are **private**, access to attributes goes through the available methods

2) Information hiding: Internal implementation is hidden from external code, only the **public** interface of the class is accessible
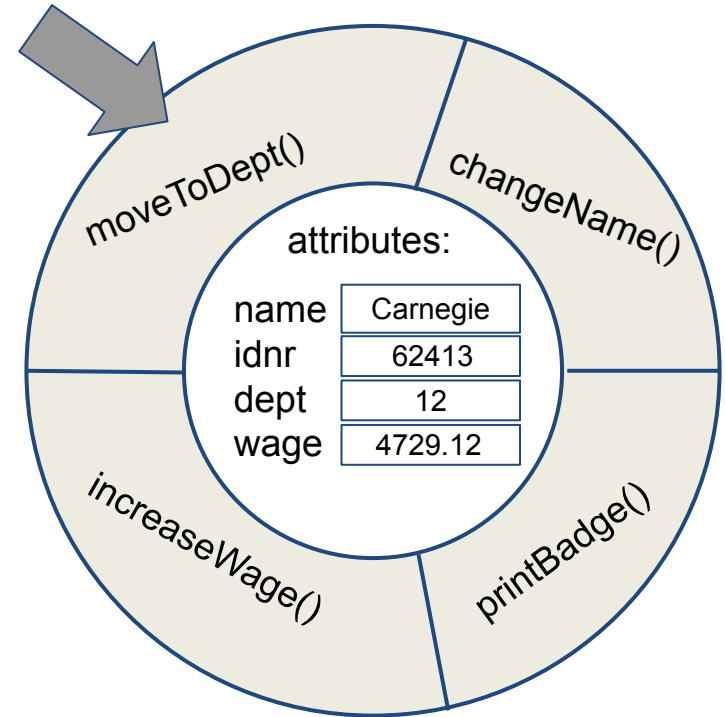
6.1. Classes and Objects

Access to object's attributes goes through the available methods. Example:

```
GPSCoord here; // GPS coordinate object

// we can modify latitude and
// longitude only together:
here.set(50.883849, 8.020959);

// we can modify the elevation:
here.setElevation(285.128);
// we can retrieve these attributes:
double lat = here.getLatitude();
double lng = here.getLongitude();
// .. but not elevation
```


attributes:

lat	50.883849
lng	8.020959
elev	285.128

getLatitude()   getLongitude()   set()   setElevation()
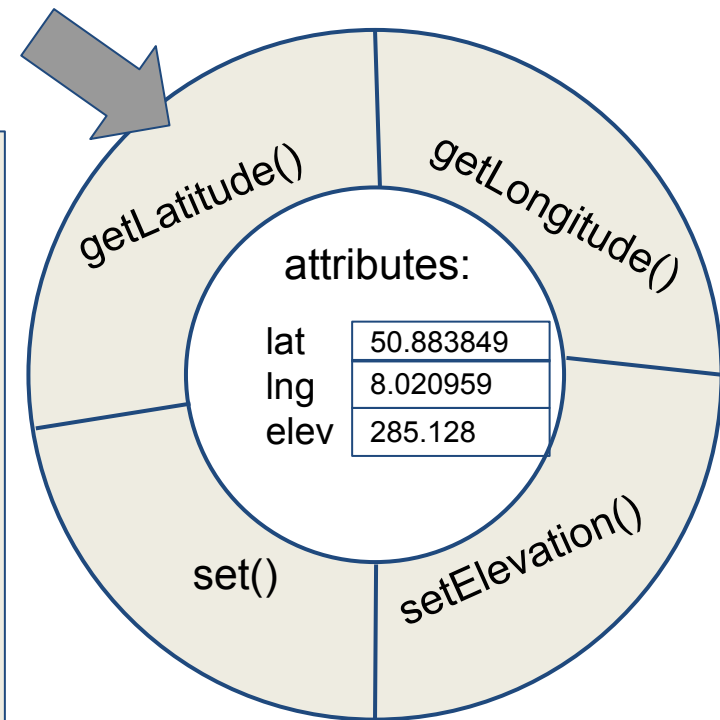
6.1. Classes and Objects

Declaring a class GPSCoord:

```cpp
// GPS coordinate class:
class GPSCoord {
 private:
  // latitude, longitude, elevation:
  double lat, lng, elev;
 public:
  // set latitude and longitude:
  void   set(double la, double lo);
  void   setElevation(double val);
  double getLatitude();
  double getLongitude();
};
```


attributes:

lat	50.883849
lng	8.020959
elev	285.128

getLatitude() getLongitude() set() setElevation()
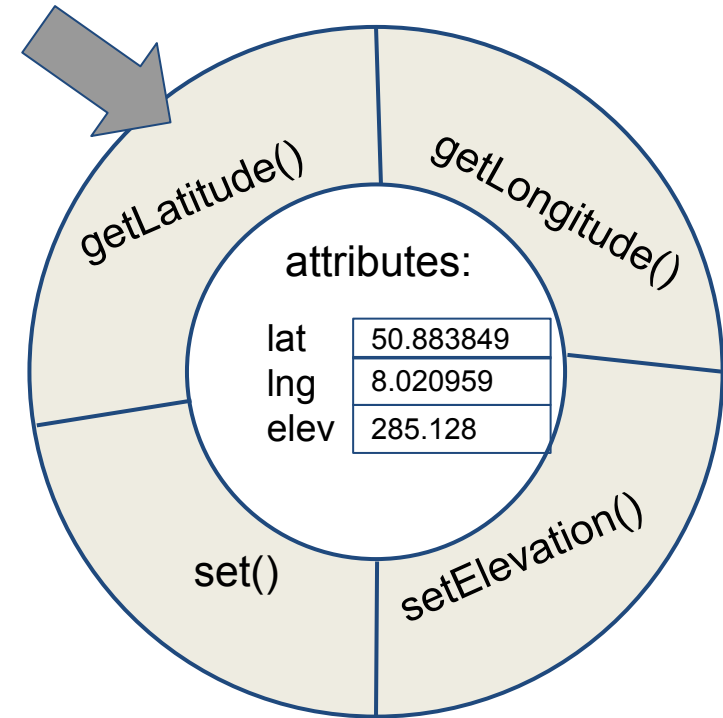
6.1. Classes and Objects

Implementing the methods for the class GPSCoord:

```cpp
void GPSCoord::set(double la, double lo){
  lat = la; lng = lo;
}
void GPSCoord::setElevation(double val){
  elev = val;
}
double GPSCoord::getLatitude(){
  return lat;
}
double GPSCoord::getLongitude(){
  return lng;
}
```

6.1. Classes and Objects

Creating an object of the class GPSCoord:

```
GPSCoord here; // GPS coordinate object

// we can modify latitude and
// longitude only together:
here.set(50.883849, 8.020959);

// we can modify the elevation:
here.setElevation(285.128);
// we can retrieve these attributes:
double lat = here.getLatitude();
double lng = here.getLongitude();
// .. but not elevation
```

Memory

lat	50.883849
long	8.020959
elev	285.128

set()   setElevation()
getLatitude()
getLongitude()

here
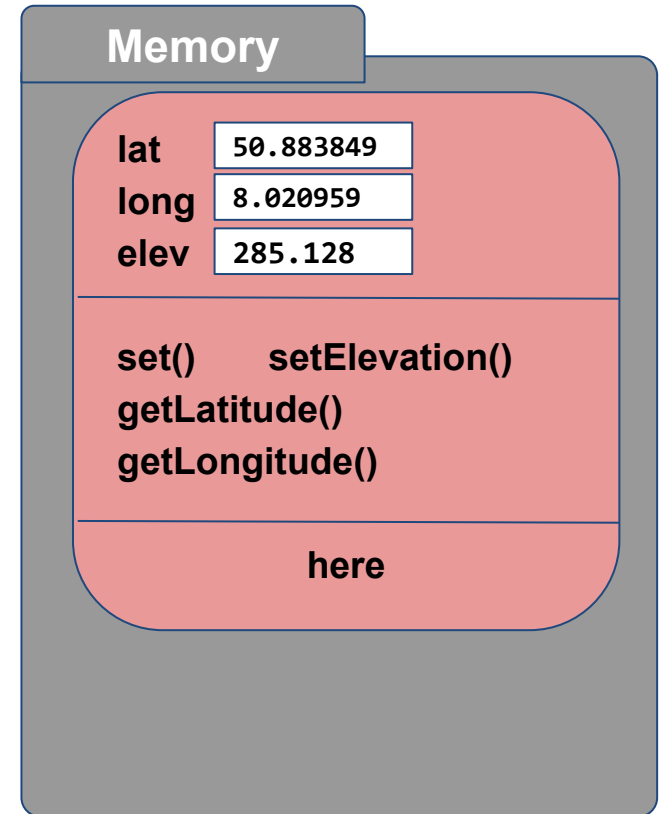
6.1. Classes and Objects

An object's method (member function) has access to *all* its attributes (variables) and methods

```cpp
void GPSCoord::set(double la, double lo){
  lat = la; long = lo;  // defaults
  // call GPSCoord methods to update:
  lat = getLatitude();
  lng = getLongitude();
  setElevation(285.128);
}
```

```cpp
GPSCoord here; // GPS coordinate object
// set latitude and longitude:
here.set(50.883849, 8.020959);
```

Memory

lat	50.883849
long	8.020959
elev	285.128

set()   setElevation()
getLatitude()
getLongitude()

here

| la | 50.883849 |
| lo | 8.020959 |

set
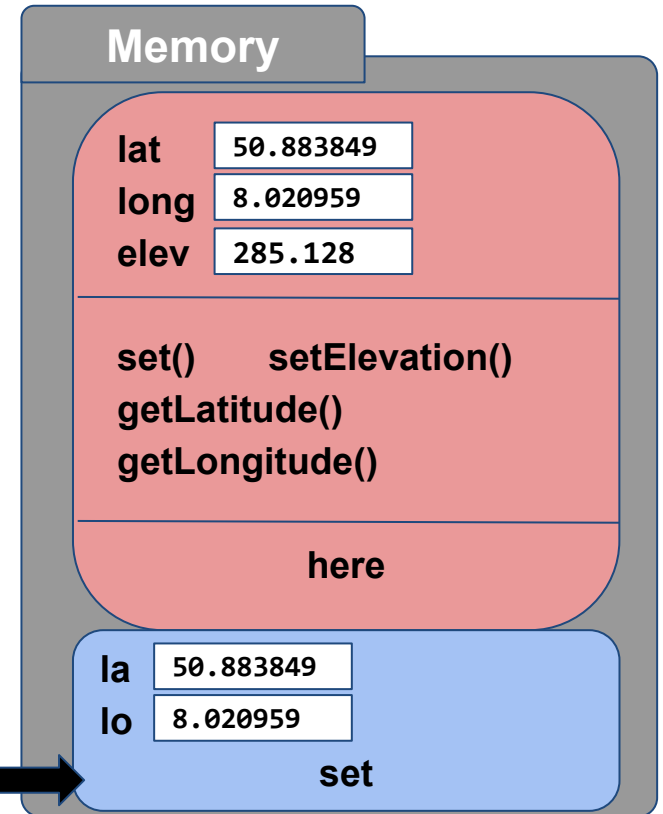
6.1. Classes and Objects

❗ Note new suggested indentation & syntax for classes. One space should come before private or public, and a colon (':') after these keywords. Example:

```cpp
// GPS coordinate class:
class GPSCoord {
 private:
  // latitude, longitude, elevation:
  double lat, lng, elev;
 public:
  // set latitude and longitude:
  void set(double la, double lo);
  void etElevation(double val);
  double getLatitude();
  double getLongitude();
};  // mind the semicolon after the declaration
```

6.1. Classes and Objects: Minor notes

Attributes could also be public (but usually aren't)
Methods can also be implemented in the class declaration

```cpp
class Test {
 private:
  int attribute1;  // a private attribute
 public:
  bool attribute2;  // a public attribute
  void method1(int parameter) { attribute1 = parameter; };   // methods implemented
  void method2() { std::cout << attribute1 << std::endl; };  // in class declaration
};

int main(){
  Test myTest;  // create an object of class Test
  myTest.attribute2 = false;  // we can access public attributes
  myTest.method1(21); // we can call public methods
}
```

6.1. Classes and Objects: Minor notes

The class declaration is usually in the file `className.h`, the class' method implementations in the file `className.cpp`

```
class Test {                    Test.h
 private:
  int attribute1;
 public:
  bool attribute2;
  void method1(int parameter);
  void method2();
};
```

```
#include "Test.h"              Test.cpp
void Test::method1(int parameter) {
  attribute1 = parameter;
};
void Test::method2() {
  std::cout << attribute1 << std::endl;
};
```

```
#include "Test.h"              mainTest.cpp
int main(){
  Test myTest;
  myTest.attribute2 = false;
  myTest.method1(21);
}
```

6.1. Classes and Objects: Minor notes: `struct`

`struct` allows to group multiple variables in C++ into one variable. All of its attributes (or members) and methods (or member functions) are public.
Since C++20, `struct`s (as well as `class`es) can use **Designated Initializers**.

```cpp
struct Key {
 char label;
 bool pressed;
};

int main() {
  Key item1;
  item1.label = 'q';
  item1.pressed = false;
  // initialize members to values:
  Key item2 = {'w', true};
}
```

```cpp
struct Date {
 int day, month, year;
};
int main() {
  // Designated Initializers allow to
  // make initialization more flexible
  // (from C++20)
  Date today = { .day = 9, .month = 5,
                 .year = 2025
  }; // or shorter:
  Date next{.day=10,.month=5,.year=2025};
}
```

6.1. Classes and Objects: Minor notes: Scope Resolution

The scope resolution operator `::` lets you access variables from other scopes:

```cpp
#include <iostream>  // std scope

auto a = 'a';  // global scope

namespace myscope {  // myscope scope
  auto a = "a string!";
}

int main() {
  auto a = 7;
  std::cout <<  // std scope
    "a = " << a << '\n' <<   // local scope
    "::a = " << ::a << '\n' <<   // global scope
    "myscope::a = " << myscope::a << '\n';  // "myscope" scope
}
```

6.2. Constructors and Destructors

Reminder: Variables can be declared and later initialized, or they can be immediately initialized within the declaration:

```cpp
char mySymbol = '?';
```

This declaration and initialization can be done for classes' objects as well:

```cpp
GPSCoord myLocation(50.88385, 8.02096, 285.128);  // coordinates of Siegen

Employee user("Carnegie", 62413, 12, 4729.12);  // employee Carnegie

SizedSymbol bigQuestion('?', 14);  // a SizedSymbol '?' with size 14
```

This requires a special method with the class' name: The constructor

6.2. Constructors and Destructors

A constructor has no return value (not even void) and is automatically called

```cpp
class Test {
 private:
  int attribute1;
 public:
  Test(int parameter) {  // this is a constructor for class Test
    attribute1 = parameter;
  };
  void method2() {
    std::cout << attribute1 << std::endl;
  };
};

int main(){
  Test myTest(21);    // object's constructor initializes is automatically called
  myTest.method2();  // this will print out '21' to the terminal
}
```

6.2. Constructors and Destructors

Constructors can be overloaded (distinguished by their parameters):

```cpp
class Test {
 private:
  int attribute1, attribute2;
 public:
  // multiple constructors for class Test:
  Test() { attribute1 = 0; }; // this is a default constructor
  Test(int parameter) { attribute1 = parameter; };
  Test(int parameter1, int parameter2) { attribute1 = parameter2; };
  void method2() { std::cout << attribute1 << std::endl; };
};

int main(){
  Test myTest(4, 12);  // object of class Test has attribute1 initialized to 12
  myTest.method2();  // this will print out '12' to the terminal
}
```

6.2. Constructors and Destructors

- A class' default constructor is a constructor without parameters
- A class without declared constructors results in the compiler automatically generating a default constructor
- A default constructor is invoked when an object is created, but not initialized

```cpp
class Test {
 private:
  int attribute1;
 public:
  // Test() {}  --> an automatically generated constructor would look like this
  void method2() { std::cout << attribute1 << std::endl; };
};

int main(){
  Test myTest;  // this object is initialized with empty default constructor
}
```

6.2. Constructors and Destructors

A destructor is automatically called whenever an object is destroyed:

```cpp
bool myFunction(){
  Test myTest(17);   // this object is created and initialized here
  return false;      // myTest's destructor is called when function returns
}
```

This destructor is another special method with the same name as the class, starting with a ~:

```cpp
Test::~Test() {  // this is the destructor for class Test
  // here come statements for application-specific clean up
}
```

6.2. Constructors and Destructors

Example 00 (difficulty level: 🌶️)

```cpp
/**
  Write a program that declares, implements, and uses a class with no attributes.
  The class should print 'hello' to the terminal when its object is created and
  'bye' when its object is removed from memory.
*/
#include <iostream>      // terminal input and output classes and objects

// write the class here

int main() {
  // create a class object here

}
```

6.2. Constructors and Destructors

Example 01 (difficulty level: 🌶️🌶️)

```
/**
  Write a program that declares, implements, and uses a class with two attributes,
  a boolean called 'flag' and an integer called 'number', which can only be changed
  or read through a constructor. The class should also have a method 'get' with no
  parameters, which returns the integer 'number' only if 'flag' is true, and
  otherwise 0.
*/
// write the class here
int main() {
  int returnValue;
  // create a class object here
  // and use its get method
  return returnValue;
}
```

6.2. Constructors and Destructors: Maze Game v.4.00

Change the module into a class Maze, with `mazeGame.cpp` looking this way:

```cpp
/* Fourth draft of Maze Game: drawing is in class "Maze" */    mazeGame.cpp
#include "Maze.h"   // everything related to the maze
int main() {
  auto c = ' ';         // used for user key input
  Maze maze(10, 5);    // initialize the maze and put player at (10, 5)
  while ( c != 'q' ) {  // as long as the user doesn't press q ..
    maze.draw('@', 3);    // draw player as a '@' with color pair 3
    c = getch();          // capture the user's pressed key
    switch (c) {
      case 'w': maze.up(); break;  // go up
      case 's': maze.down(); break;  // go down
      case 'a': maze.left(); break;  // go left
      case 'd': maze.right(); break;  // go right
    }
  }
}
```

6.3. `this` and initializing `const` attributes

Class methods often reuse attribute names, leading to a problem:

```cpp
class Maze {
 public:
  Maze(int16_t x, int16_t y);
  …
 private:
  int16_t x, y;
  …
};
```

```cpp
Maze::Maze(int16_t x, int16_t y) {
  // how to assign the values of
  // constructor parameters x and y
  // to class attributes x and y?
  …
}
Maze::Maze() {
  int16_t x, int16_t y;
  // how to assign the values of
  // constructor parameters x and y
  // to local variables x and y?
  …
}
```

6.3. `this` and initializing `const` attributes

One solution that works in all the class' methods is to use **this**

```
Maze::Maze(int16_t x, int16_t y) {
  this->x = x;  // attribute x = value of constructor parameter x
  this->y = y;  // attribute y = value of constructor parameter y
  ...
}
```

- Note: Each object gets its own copy of data members, all objects share a single copy of the class' methods
- `this` is an implicit pointer (see later) that is passed as a hidden parameter for all the class' methods, and is there available as another local variable

6.3. `this` and initializing `const` attributes

`this` is passed as a hidden parameter for all the class' methods, as an extra (pointer) variable

```cpp
void GPSCoord::set(double lat, double lng){
  this->lat = lat;
  this->lng = lng;
  this->lat = getLatitude();
  this->lng = getLongitude();
  setElevation(285.128);
}
```

```cpp
GPSCoord here; // GPS coordinate object
// set latitude and longitude:
here.set(50.883849, 8.020959);
```

Memory

lat	50.883849
lng	8.020959
elev	285.128

set()   setElevation()
getLatitude()
getLongitude()

here

la	50.883849	this
lo	8.020959

set
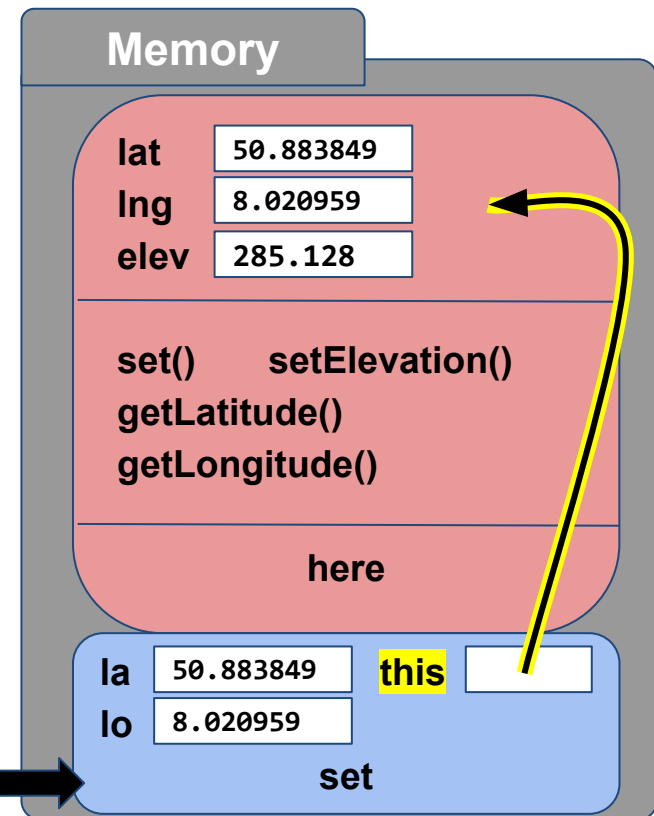
6.3. **this** and initializing **const** attributes

Another (shorter) solution for constructors is to use this initialization syntax:

```cpp
Maze::Maze(int16_t x, int16_t y) : x(x), y(y) {
 // attributes x and y have now the same value as constructor
 // parameters x and y
}
```

This *member initializer list* syntax in constructors is not an assignment but a real initialization of the attribute. Curly braces can also be used:

```cpp
Maze::Maze(int16_t x, int16_t y) : x{x}, y{y} {
 // attributes x and y have now the same value as constructor
 // parameters x and y
}
```

6.3. `this` and initializing `const` attributes

This syntax addresses the problem of initializing a class' `const` attribute:

```cpp
class Maze {
  ...
 private:
  const int16_t mazeXlen;
  const int16_t mazeYlen;
  ...
};
```

```cpp
Maze::Maze(int16_t x, int16_t y) {
    // we cannot assign to const:
    mazeXlen = 15; // compiler error
    mazeYlen = 10; // compiler error
    ...
}
```

This syntax is not an assignment but an initialization, and thus works:

```cpp
Maze::Maze(int16_t x, int16_t y) : mazeXlen(15), mazeYlen(10) {
  // mazeXlen is now 15, mazeYlen 10

  ...
}
```

6.3. `this` and initializing `const` attributes

This syntax similarly addresses other initialization problem, e.g. for C strings:

```cpp
class Label {
  ...
 private:
  // name is initialized:
  char name[25] = "default";
  ...
};
```

```cpp
Label::Label() {
  name = "none";  // --> compiler error
}
Label::Label(int8_t i) {
  name = "int8_t"; // --> compiler error
}

  ...
```

This syntax is not an assignment but an initialization, and thus works:

```cpp
Label::Label() : name("none") { /* name is now "none" */ }
Label::Label(int8_t i) : name("int8_t") { /* name is now "int8_t" */ }

  ...
```

6.4. `static` members

- Each object has its own class attributes
- All objects share one copy of the class' methods
- **`static`** attributes, or [static members](#), are stored once across all objects. *Changing it through one object will change it for all objects.*

```
double GPSCoord::prec;  // define static member once
GPSCoord here;
GPSCoord there;

there.prec = 0.02;
// the following will print out 0.02:
std::cout << here.prec;
```

Memory

lat	50.883849
lng	8.020959
elev	285.128

GPSCoord here

static prec `0.02`

lat	50.883845
lng	8.020939
elev	285.121

GPSCoord there
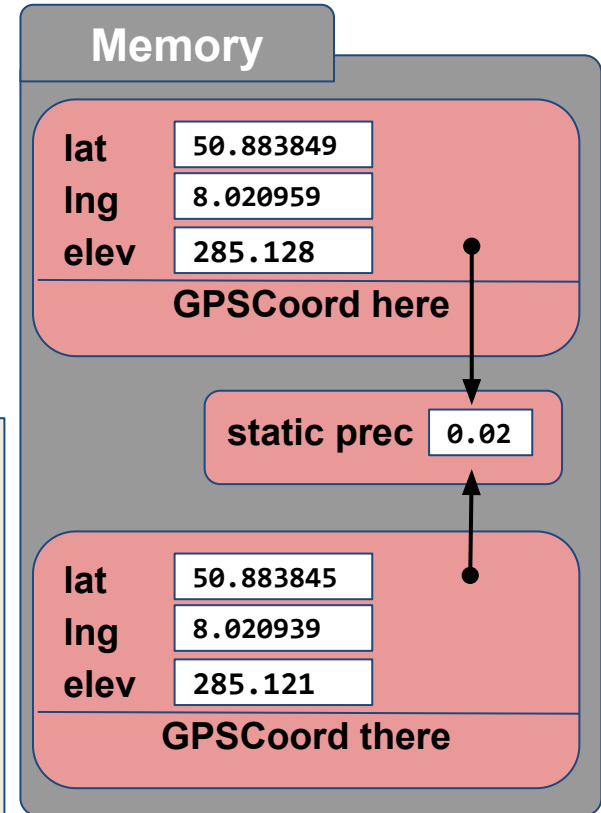
6.4. `static` members

- They exist even if no objects of the class have been defined
- Static class members are declared in the class declaration, and are defined (and/or initialized) in the implementation / source file:

Memory

lat	50.883849
Ing	8.020959
elev	285.128

GPSCoord here

static prec `0.02`

lat	50.883845
Ing	8.020939
elev	285.121

GPSCoord there

GPSCoord.h

```cpp
class GPSCoord {
 public:
  // declaring a precision
  // attribute for all
  // GPSCoord objects:
  static double prec;
  …
};
```

GPSCoord.cpp

```cpp
// precision definition
// note that only here
// the variable is
// in fact created:
double GPSCoord::prec;

…
```
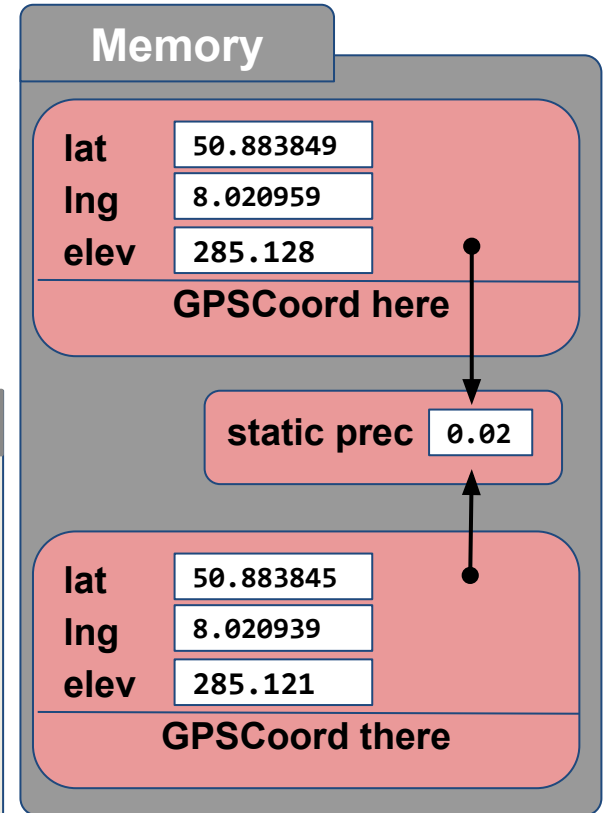
6.4. `static` members

These are not the same as `static` variables:

- Local variables that are declared as `static` are stored as global variables in the static data segment in memory
- Their size has to be known at compile time (e.g., for arrays)

```cpp
void GPSCoord::set(){
    // this variable will stay in memory and keep
    // its value after the method finishes:
    static double elevationPrec = 0.01;
    setElevation(285.128, elevationPrec);
}
```

Memory

lat	50.883849
lng	8.020959
elev	285.128

GPSCoord here

static prec `0.02`

lat	50.883845
lng	8.020939
elev	285.121

GPSCoord there
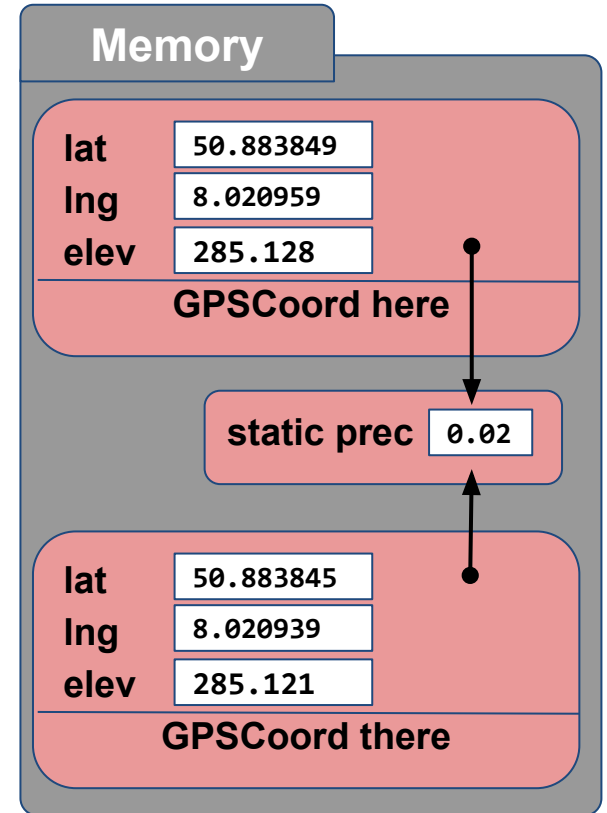
6.4. `static` members

Example 02 (difficulty level: 🌶️)

```cpp
/**
  Create a class with just a static member, and illustrate that it exists, even
  without any objects.
*/
#include <iostream>    // terminal output
// write the class below:

int main() {
  // do not create an object here, but assign a value to static member here
  std::cout << " The value of the static member is ";
  // print its value here:

  std::cout << '\n';
}
```

6.4. `static` members

Example 03 (difficulty level: 🌶️🌶️)

```cpp
/**
  Create a class Counter below with a static member "count" that holds the number of
  objects of that class. Also, the public attribute "type" of the class should
  contain the type of the supplied parameter to the constructor, as a C string.
*/
#include <iostream>      // terminal output
// write the class, and then define the static member below:


int main() {
  Counter c1, c2(2), c3(3.4);
  // The next line should return "void,int,double,3":
  std::cout << c1.type << ',' << c2.type << ',' << c3.type << '\n';
  // Finally, print out the count of Counter objects below:

}
```

6.5. The **string** Class

iostream has a class called <u>**std::string**</u>, helping in dealing with strings:

```cpp
#include <iostream>       // terminal input and output classes and objects
int main() {
  std::string myFirstName("John");  // initialize with constructor
  std::string myLastName = "Doe";   // initialize with assignment
  std::string myString = myFirstName + myLastName;  // concatenation

  std::cout << myString << ", length = " ;
  std::cout << myString.length() << '\n';  // returns myString length

  std::cout << " Do found at = ";
  std::cout << myString.find("Do") << '\n';  // return position of "Do"

  std::cout << myString.compare(4, 3, "Do");  // is substring at position 4
  std::cout << '\n';                           // and length 3 the same as "Do"?
}
```

6.5. The **string** Class

iostream has a class called <u>std::string</u>, helping in dealing with strings.
Several **std::string** alternatives use instead of **char**:

<u>std::wstring</u>   uses **wchar_t** for <u>wide strings</u>
std::u8string (since C++20)   uses **char8_t**
std::u16string (since C++11)   uses **char16_t**
std::u32string (since C++11)   uses **char32_t**

```cpp
#include <iostream>      // terminal input and output classes       strings.cpp
int main() {
  std::wcout.imbue(std::locale("en_US.UTF-8"));
  std::string  myStr  = "¡Hola! 日本 שלום 你好 مرحبا";    // UTF-8
  std::wstring mywStr = L"¡Hola! 日本 שלום 你好 مرحبا";   // wide chars
  std::cout  << myStr << '\n' << sizeof(myStr[0]) << '\n';
  std::wcout << mywStr << '\n' << sizeof(mywStr[0]) << '\n';  // ! note wcout
  // what do sizeof(myStr) and sizeof(mywStr) return? Try it out and explain
}
```

6.6. Operator Overloading

Classes can redefine operators for their objects. For example, the operator **+** in a String class can be implemented to concatenate two strings, or it can be implemented in the class Complex to add two complex numbers:

```cpp
#include <iostream>                                         // Complex.cpp

class Complex {
 private:
  int real, imag;
 public:
  Complex(int real=0, int imag=0): real(real), imag(imag) {}  // sole constructor
  Complex operator +(Complex const & obj) {  // takes the object on right of '+'
    Complex res;                              // this object is on the left of '+'
    res.real = real + obj.real;
    res.imag = imag + obj.imag;
    return res;
  }
```

6.6. Operator Overloading

- Note that **conversion** can also be used as an operator, *these have no return type*
- Any constructor works as **conversion constructor** (since c++11)
- A **default assignment** operator (=) is automatically made for each class

```cpp
operator std::string() {  // conversion to std::string    Complex.cpp
  return std::to_string(real)+'+'+std::to_string(imag)+'i';
}
operator float() {  // conversion to float
  return (float)real+(float)imag/1000.0f;
}
};  // end of Complex class

int main() {
  Complex c1(2,3), c2(7,5), c3(3,3);
  std::cout << (std::string)(c1 + c2) << '\n';  // out: 9+8i
  c1 = 6; c2 = {7, 8}; c3 = c2;  // conversion constructor, default assignment
  std::cout << c1 << ' ' << c2 << ' ' << c3 << '\n';  // out: 6 7.008 7.008
}
```

6.6. Operator Overloading

- The default assignment is defined to operate recursively - assigning each data element according to the appropriate rules for the type of that element. Each member of a class type is assigned by calling that member's assignment operator. Members that are of built-in type are assigned by assigning their values. [Accelerated C++, Koenig, Hendrickson, Moo]

- Some operators *cannot* be overloaded, such as: `sizeof`, scope resolution (`::`), class member access (or dot: `.`) and the ternary operator (`? :`)

- Recommended for more information on operator overloading:

  https://stackoverflow.com/questions/4421706/what-are-the-basic-rules-and-idioms-for-operator-overloading

6.6. Operator Overloading

Example 04 (difficulty level: 🌶️🌶️)

```cpp
#include <iostream>      // terminal input and output classes and objects
class GPSCoord {
 private:
  double lat, lng, elv;  // latitude, longitude, elevation
 public:
  GPSCoord(double lat, double lng, double e=0.0): lat(lat), lng(lng), elv(e) {}
  operator std::string() {
    // convert the three attributes to a string
  }
  bool operator == (GPSCoord & obj) {
    // test whether two objects have same latitude and longitude
  }
};
int main() {
  GPSCoord point1(1.0,2.0,3.0), point2(1.0,2.0,4.0);
  // print out both points, and whether they overlap with the == operator
}
```

6.6. Operator Overloading

Example 05 (difficulty level: 🌶️🌶️)

```cpp
#include <iostream>      // terminal input and output classes and objects
class GPSCoord {
 private:
  double lat, lng, elv;  // latitude, longitude, elevation
 public:
  GPSCoord(double lat, double lng, double e=0.0): lat(lat), lng(lng), elv(e) {}
  int operator [] (int index);  // access the GPS coordinates as index
};
int GPSCoord::operator [] (int index) {
  // for index 0, 1 or 2, return the latitude, longitude, or elevation
}
int main() {
  GPSCoord point1(1.0,2.0,3.0);
  // print out the coordinates by accessing them with the [] operator

}
```

6.7. The `ifstream` and `ofstream` Classes

The module **fstream** contains a class **std::ifstream** for reading from a file:

```cpp
#include <iostream>     // terminal input/output classes & objects
#include <fstream>      // input file stream class std::ifstream

int main() {
  std::ifstream myFile("fileTest.cpp");  // initialize with constructor

  auto c = ' ';
  // get a character from the file, move to next, and output it to the terminal
  while (myFile.get(c)) std::cout << c;
  std::cout << '\n';
}
```

`fileTest.cpp`

6.7. The `ifstream` and `ofstream` Classes

The module `fstream` also contains a class **std::ofstream** for writing to a file:

```cpp
#include <iostream>  // terminal input/output classes & objects      copyTest.cpp
#include <fstream>   // in/output file stream classes

int main() {
  std::ifstream myFile("copyTest.cpp");       // initialize input and output file
  std::ofstream myFileCopy("copyTest_copy.cpp");  // streams with constructors

  auto c = ' ';
  // get each character from input file stream and output to output file stream
  while (myFile.get(c)) myFileCopy << c;
  std::cout << '\n';
}
```

6.7. The `ifstream` and `ofstream` Classes

Both are specific input and output streams (see later), that typically use insertion (**<<**) and extraction (**>>**) operators.

Example 06 (difficulty level: 🌶️)

```cpp
#include <iostream>        // terminal input and output classes and objects
class GPSCoord {
 private:
  double lat, lng, elv;  // latitude, longitude, elevation
 public:
  GPSCoord(double lat, double lng, double e=0.0): lat(lat), lng(lng), elv(e) {}
  // add the >> operator, so that the main function works as advertised
  // on any output stream. Use as a parameter: std::ostream & out
};
int main() {
  GPSCoord point1(1.0,2.0,3.0);
  // print out the coordinates with the >> operator, followed by a newline
  point1 >> std::cout;
}
```

6.8. The `tuple` Class

A `std::tuple` is a fixed-sized collection of values of various data types (preview of what is still to come, as it uses templates under the hood):

```cpp
#include <iostream> // for std::cout and tuples functionality
int main() {
  auto myUser = std::make_tuple("James", "Smith", 187.2);  // auto → std::tuple
  // get with index-based access:
  std::cout << std::get<0>(myUser) << " " << std::get<1>(myUser);
  // get with type-based access:
  std::cout << ":" << std::get<double>(myUser) << '\n';
  // output: "James Smith: 187.2" and goes to next line
}
```

- `get<0>(myUser)` accesses first element (hence index-based access, since C++11)
- `get<double>(myUser)` accesses the double (hence type-based access, since C++14) (works only if 1 tuple element has this type, otherwise the compiler reports an error)

6.8. The `tuple` Class

With decomposition declarations or <u>structured bindings</u> (since C++17), you can unpack the contents of the tuple into individual variables:

```cpp
#include <iostream> // for std::cout and tuples functionality
int main() {
  auto myUser = std::make_tuple("James", "Smith", 187.2);  // auto → std::tuple
  auto [fname, lname, height] = myUser;  // decomposition declaration, C++17
  std::cout << fname << " " << lname << ":" << height << '\n';
}
```

the first **auto** above can deduce myUser as an **std::tuple** object, as that is the return type from **std::make_tuple()**

6.8. The `tuple` Class

One use of `std::tuple` is to return such an object from a function, bundling several values (even of different types) together:

```cpp
#include <iostream> // for std::cout and tuples functionality

auto arithmetic(int a, int b) {  // auto here is possible since C++14
  return std::make_tuple( a + b, a - b, a * b, a / b );
}

int main() {
  auto arith = arithmetic(5, 7);  // arithmetic returns a 4-int tuple
  std::cout << " add: " << std::get<0>(arith) << '\n';
  std::cout << " sub: " << std::get<1>(arith) << '\n';
  std::cout << " mul: " << std::get<2>(arith) << '\n';
  std::cout << " div: " << std::get<3>(arith) << '\n';
}
```

6.8. The `tuple` Class

One use of `std::tuple` is to return such on object from a function, bundling several values (even of different types) together:

```cpp
#include <iostream> // for std::cout and tuples functionality

auto arithmetic(int a, int b) {  // auto here is possible since C++14
  return std::make_tuple( a + b, a - b, a * b, a / b );
}

int main() {  // now using structured bindings, since C++17
  auto [add, sub, mul, div] = arithmetic(5, 7);
  std::cout << " add: " << add << '\n';
  std::cout << " sub: " << sub << '\n';
  std::cout << " mul: " << mul << '\n';
  std::cout << " div: " << div << '\n';
}
```

7.1. Pointers

7.2. `new` and `delete`

7.3. Creating and deleting objects

7.4. Pointers and arrays

7.5. References

7.6. Call-by-Reference

7.7. Copy Constructors

7.8. `const` and `const` Pointers

7.9. Passing functions to functions

7.10. Smart Pointers in C++

7.1. Pointers

Reminder: Variables and objects reside in memory. Through the variable name, you can read and change the variable's value. Its type tells the compiler how.

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

Memory

myInteger=int

| 0000 0000 | 0000 0000 | 0000 0000 | 0000 1100 |

mySymbol=char

| 0000 0000 |

myFloat=float

| 0100 0010 | 0100 0000 | 0000 0000 | 0000 0000 |

myBoolean=bool

| 0000 0001 |
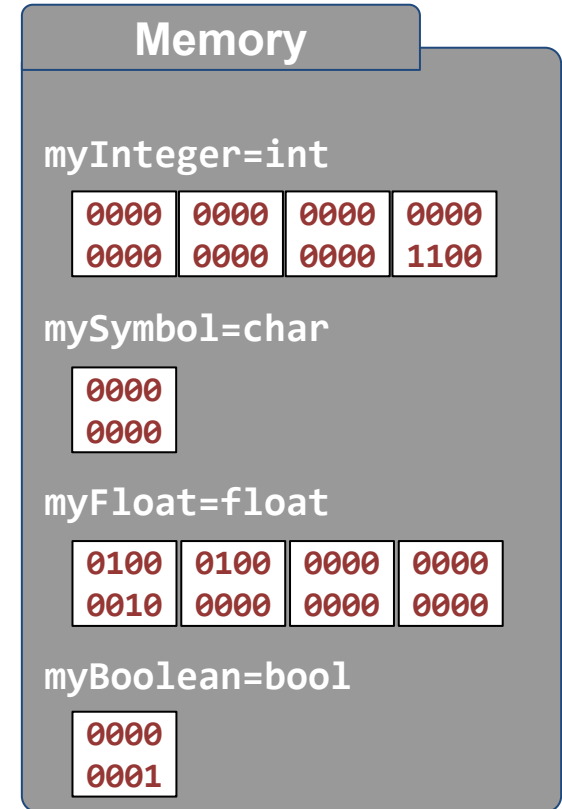
7.1. Pointers

Each memory location has a **memory address**
We assume here that one memory block occupies
one byte.

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

Memory

myInteger=int

| 0 | 0000 0000 | 0000 0000 | 0000 0000 | 0000 1100 |

mySymbol=char

| 4 | 0000 0000 |

myFloat=float

| 5 | 0100 0010 | 0100 0000 | 0000 0000 | 0000 0000 |

myBoolean=bool

| 9 | 0000 0001 |

7.1. Pointers

A pointer stores a **memory address and its associated type**

Memory

myInteger=int

| 70 | → | 0000 0000 | 0000 0000 | 0000 0000 | 0000 1100 |
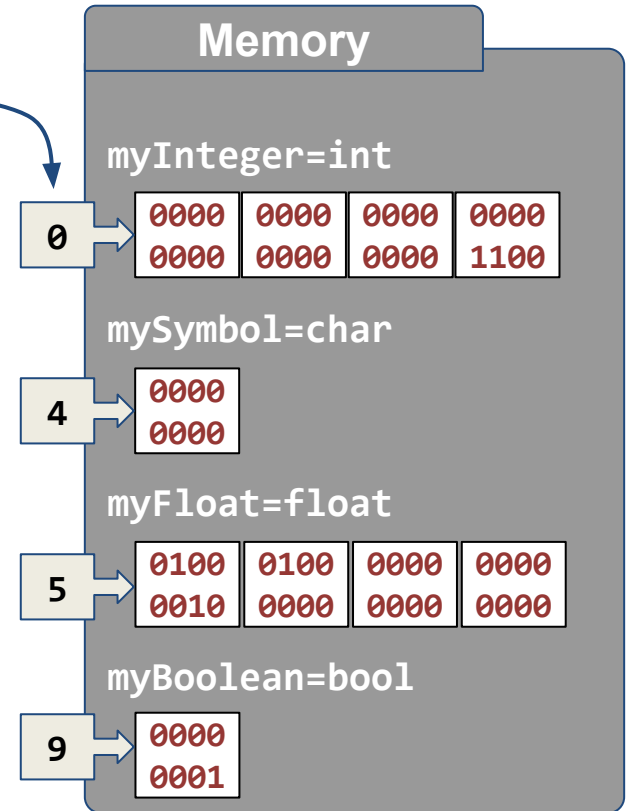
```c
/* reserving variables */
int main() {
  int myInteger = 12;    // store an integer
  int * myIntPointer;    // store a pointer to int
  myIntPointer = &myInteger;  // store floating
  *myIntPointer = 17; // myInteger is now also 17
}
```
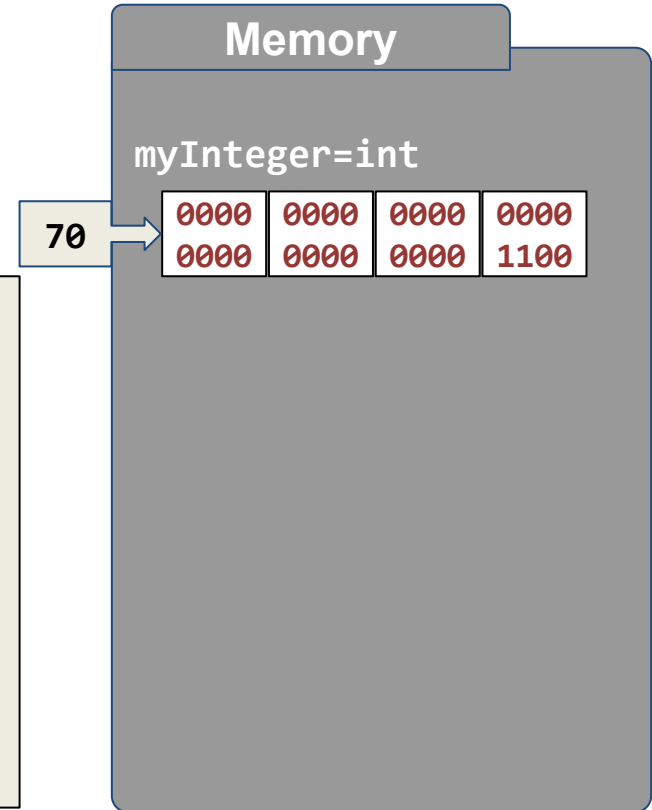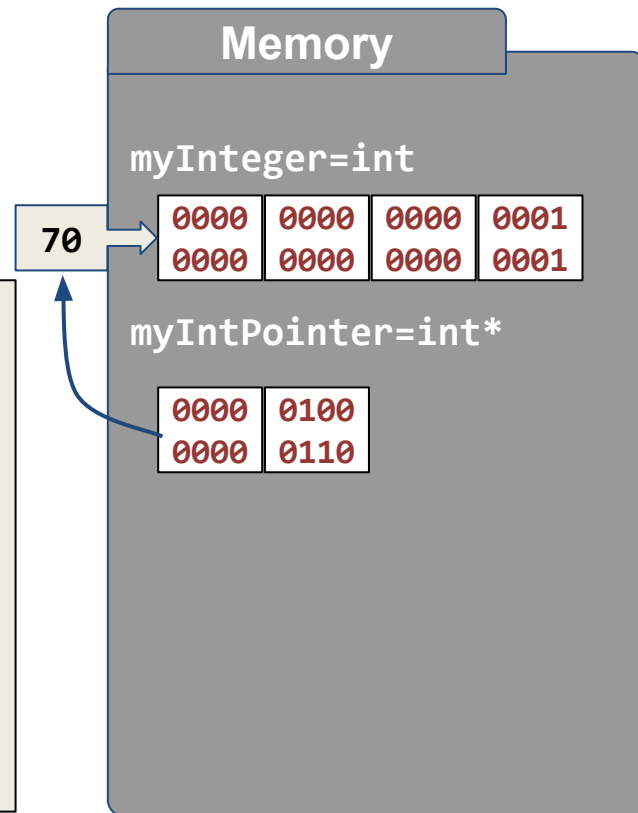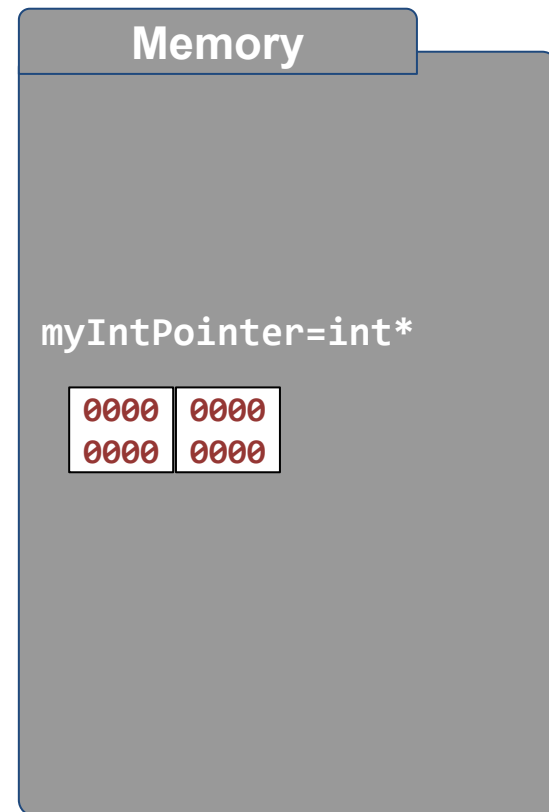
7.1. Pointers

A pointer stores a **memory address and its associated type.** Pointer variables are declared by using the ***** operator.

```
/* reserving variables */
int main() {
  int myInteger = 12;    // store an integer
  int * myIntPointer;    // store a pointer to int
  myIntPointer = &myInteger;  // store floating
  *myIntPointer = 17; // myInteger is now also 17
}
```

Memory

myInteger=int

| 0000 0000 | 0000 0000 | 0000 0000 | 0000 1100 |

70

myIntPointer=int*

| 0000 0000 | 0000 0000 |
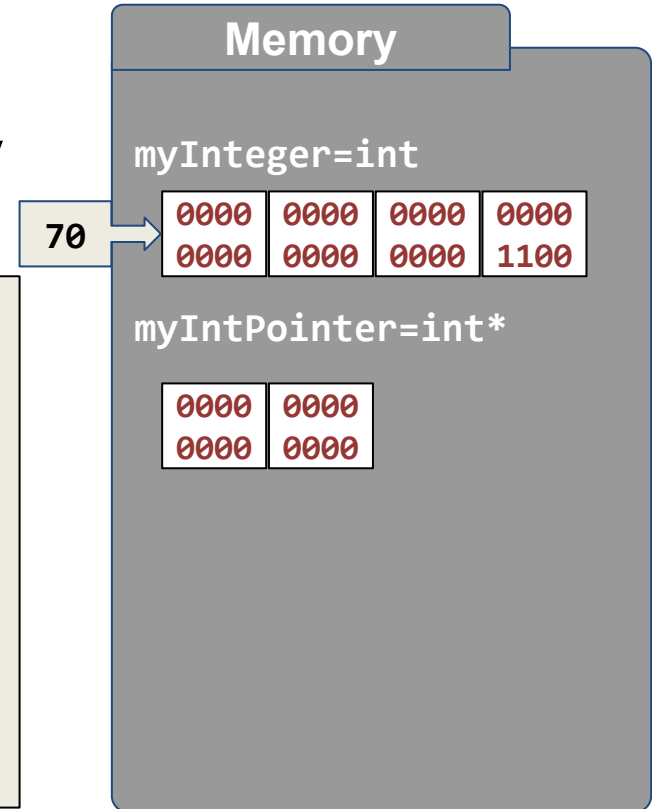
7.1. Pointers

Pointer variables can obtain the address of existing variables of that type using the **&** operator (returning the address of a variable)

```
/* reserving variables */
int main() {
  int myInteger = 12;    // store an integer
  int * myIntPointer;    // store a pointer to int
  myIntPointer = &myInteger;   // store floating
  *myIntPointer = 17; // myInteger is now also 17
}
```

Memory

myInteger=int

| 0000 0000 | 0000 0000 | 0000 0000 | 0000 1100 |

70

myIntPointer=int*

| 0000 0000 | 0100 0110 |
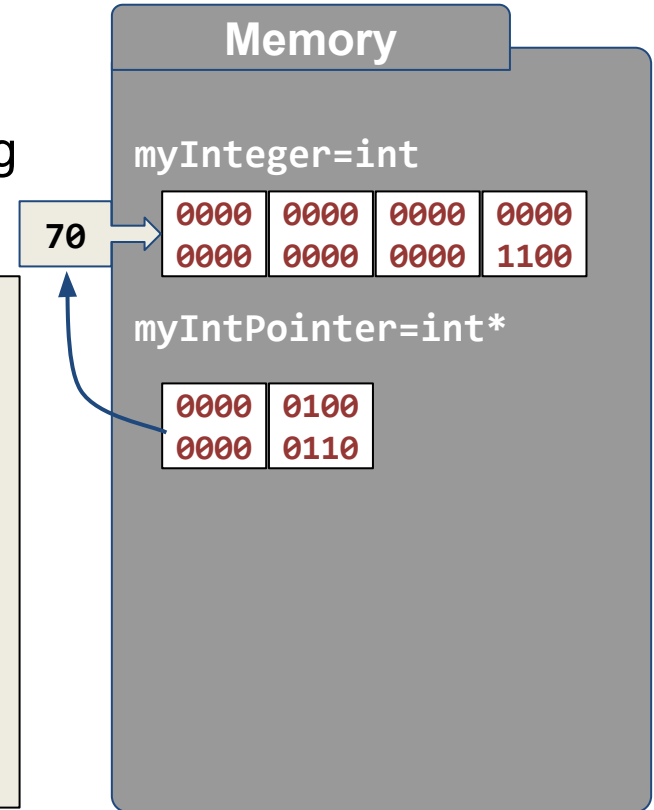
7.1. Pointers

Dereferencing a pointer means following the pointer's content (memory address) and accessing the variable of that type

```
/* reserving variables */
int main() {
  int myInteger = 12;    // store an integer
  int * myIntPointer;    // store a pointer to int
  myIntPointer = &myInteger;  // store floating
  *myIntPointer = 17; // myInteger is now also 17
}
```

Memory

myInteger=int

| 0000 0000 | 0000 0000 | 0000 0000 | 0001 0001 |

70

myIntPointer=int*

| 0000 0000 | 0100 0110 |

7.2. **new** and **delete**

A pointer assumes a memory address is reserved for a variable. Indicate that the pointer isn't pointing to a valid variable with **NULL** (or **_nullptr_** since C++11):
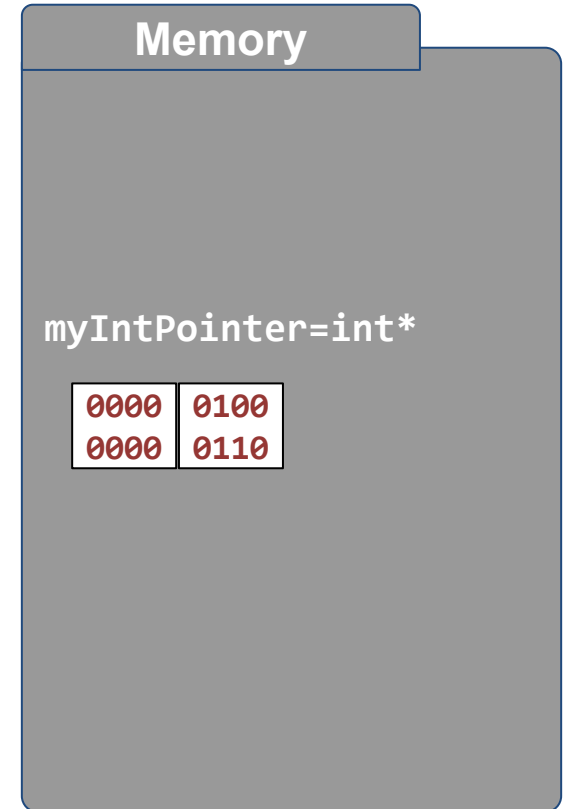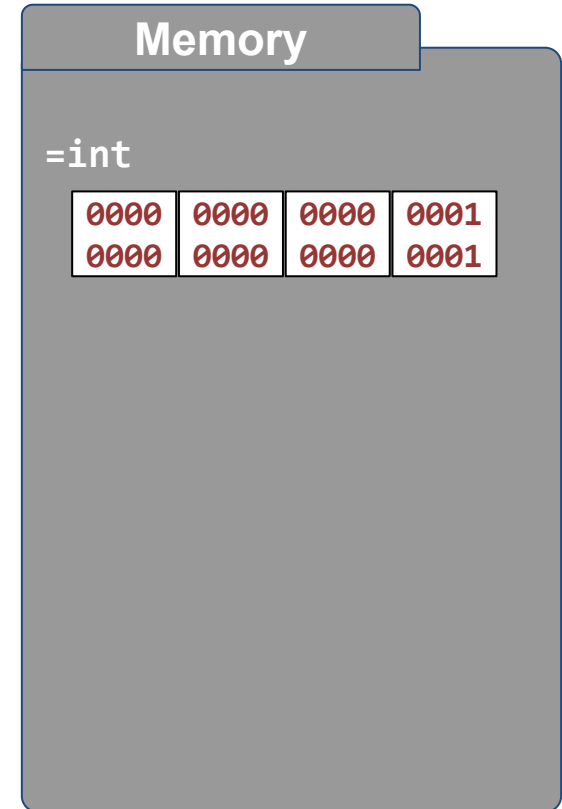
```
/* reserving variables through a pointer */
int main() {
  int * myIntPointer = NULL;  // pointer to int
  myIntPointer = new int;  // create the int
  *myIntPointer = 17;  // the int now holds 17
  delete myIntPointer;  // remove the pointer
}
```

Memory

myIntPointer=int*

0000	0000
0000	0000

7.2. **new** and **delete**

A pointer assumes a memory address is already reserved for a variable. We can create the variable of the correct type through the **new** keyword:

```cpp
/* reserving variables through a pointer */
int main() {
  int * myIntPointer = NULL;  // pointer to int
  myIntPointer = new int;  // create the int
  *myIntPointer = 17;  // the int now holds 17
  delete myIntPointer;  // remove the pointer
}
```

Memory

=int

| 0000 0000 | 0000 0000 | 0000 0000 | 0000 0000 |

70

myIntPointer=int*

| 0000 0000 | 0100 0110 |

7.2. **new** and **delete**

A pointer assumes a memory address is already reserved for a variable. This variable can only through the pointer be read and changed:

```cpp
/* reserving variables through a pointer */
int main() {
  int * myIntPointer = NULL;  // pointer to int
  myIntPointer = new int;  // create the int
  *myIntPointer = 17;  // the int now holds 17
  delete myIntPointer;  // remove the pointer
}
```

Memory

=int

0000 0000	0000 0000	0000 0000	0001 0001

70

myIntPointer=int*

0000 0000	0100 0110

7.2. **new** and **delete**

A pointer assumes a memory address is already reserved for a variable. This variable can only through the pointer be read and changed:

```cpp
/* reserving variables through a pointer */
int main() {
  int * myIntPointer = NULL;  // pointer to int
  myIntPointer = new int;  // create the int
  *myIntPointer = 17;  // the int now holds 17
  delete myIntPointer;  // remove the pointer
}
```

Memory

myIntPointer=int*

0000	0100
0000	0110

7.2. **new** and **delete**

Forgetting to **delete** a reserved variable causes a memory leak, since the pointer is removed and the variable cannot be reached anymore (but is reserved).

```cpp
void myFunction() {
  int * myIntPointer = new int; // create int
  *myIntPointer = 17;
}
int main() {
  myFunction();
  // after the above function ends, myIntPointer
  // is removed, but not the int variable
}
```

Memory

=int

0000 0000	0000 0000	0000 0000	0001 0001

7.2. **new** and **delete**

It is good practice to assign the pointer to **NULL** after **delete** (since the pointer still points to a non-reserved memory location, this is called a *dangling pointer*).

```cpp
void myFunction() {
    int * myIntPointer = new int; // create int
    *myIntPointer = 17;
    delete myIntPointer;  // remove pointer's int
    myIntPointer = NULL;  // point to NULL
}
int main() {
    myFunction();
}
```
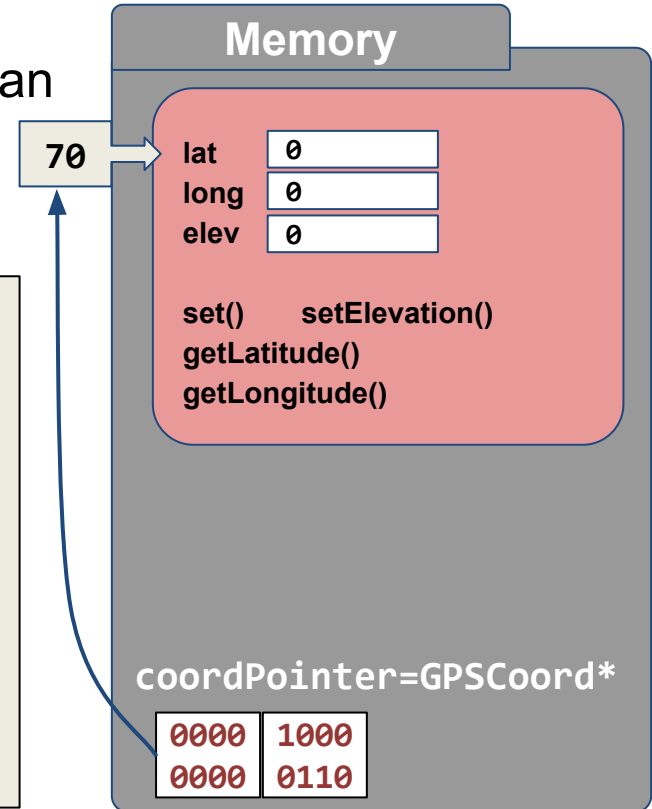
Memory

myIntPointer=int*

| 0000 | 0100 |
| 0000 | 0110 |

7.2. **new** and **delete**

It is good practice to assign the pointer to **NULL** after **delete** (since the pointer still points to a non-reserved memory location, this is called a *dangling pointer*).

```cpp
void myFunction() {
    int * myIntPointer = new int; // create int
    *myIntPointer = 17;
    delete myIntPointer;   // remove pointer's int
    myIntPointer = NULL;   // point to NULL
}
int main() {
    myFunction();
}
```

Memory

myIntPointer=int*

| 0000 | 0000 |
| 0000 | 0000 |

7.3. Creating and deleting objects

Pointers can also point to classes' objects. These can similarly be created and deleted in memory, too.

Memory

70

lat	0
long	0
elev	0

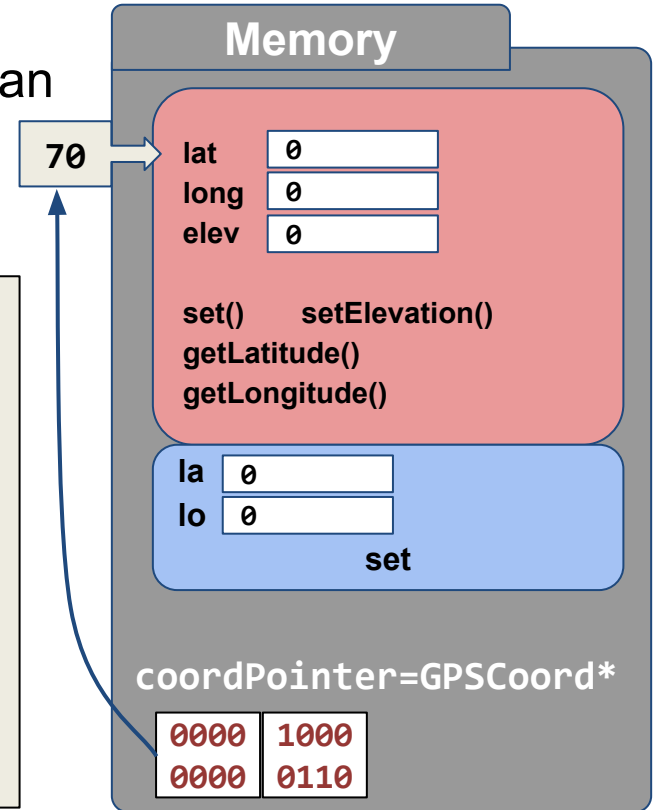set()   setElevation()
getLatitude()
getLongitude()

```cpp
void myFunction() {
  GPSCoord * coordPointer = new GPSCoord();
  coordPointer->set(0, 0);  // access method
  delete coordPointer;  // remove object
  coordPointer = NULL; // avoid dangling pointer
}
int main() {
  myFunction();
}
```
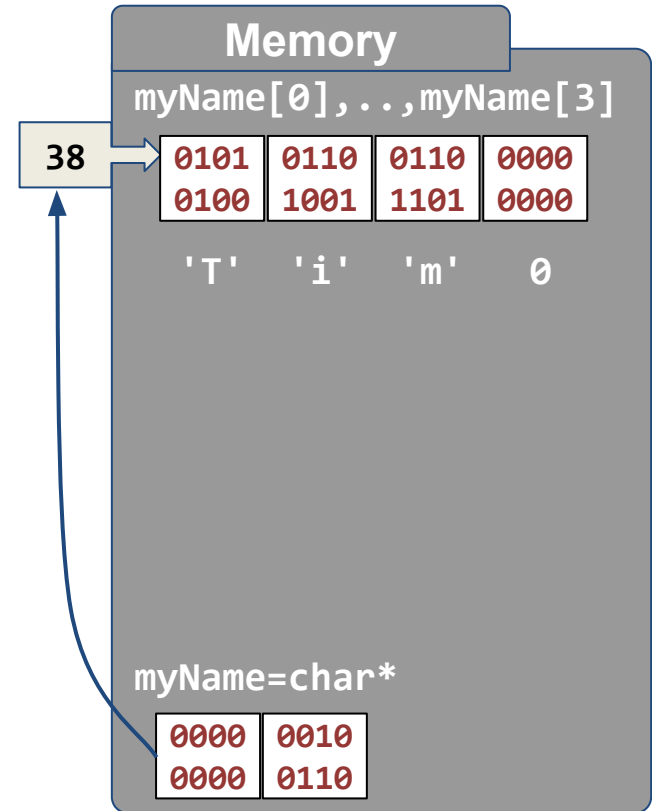
coordPointer=GPSCoord*

| 0000 | 1000 |
| 0000 | 0110 |

7.3. Creating and deleting objects

Pointers can also point to classes' objects. These can similarly be created and deleted in memory, too.

Attributes & methods are accessed with `->` operator

```cpp
void myFunction() {
    GPSCoord * coordPointer = new GPSCoord();
    coordPointer->set(0, 0);  // access method
    delete coordPointer;  // remove object
    coordPointer = NULL; // avoid dangling pointer
}
int main() {
    myFunction();
}
```

Memory

70

lat   0
long   0
elev   0

set()   setElevation()
getLatitude()
getLongitude()

la   0
lo   0
        set

coordPointer=GPSCoord*

0000 1000
0000 0110

7.3. Creating and deleting objects

Pointers can also point to classes' objects. These can similarly be created and deleted in memory, too.

Memory

70

lat | 0
long | 0
elev | 0

set()   setElevation()
getLatitude()
getLongitude()

la | 0
lo | 0

set

coordPointer=GPSCoord*

0000 | 1000
0000 | 0110
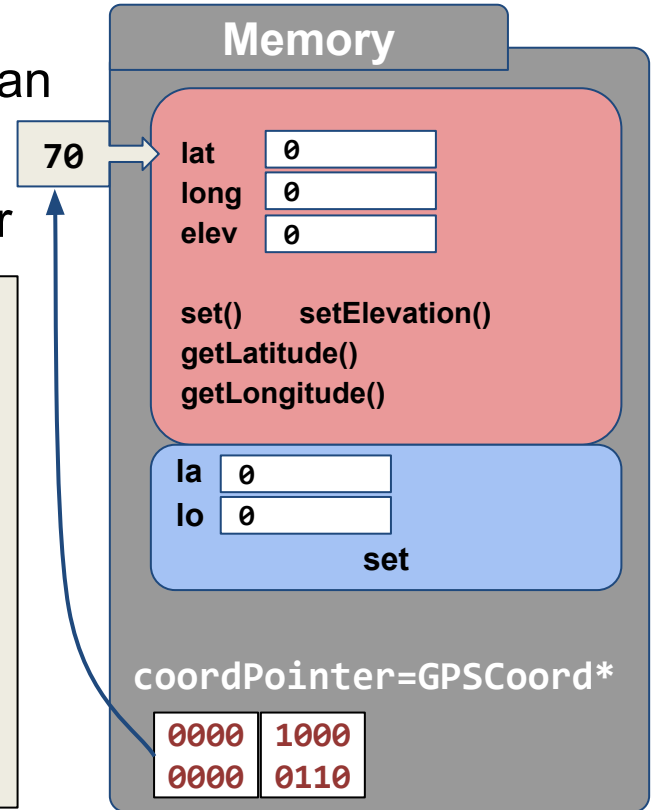
```cpp
void myFunction() {
  GPSCoord * coordPointer = new GPSCoord();
  coordPointer->set(0, 0);  // access method
  delete coordPointer;  // remove object
  coordPointer = NULL; // avoid dangling pointer
}

int main() {
  myFunction();
}
```

7.3. Creating and deleting objects

Background: Heap and stack in memory

low addresses ·· high addresses

code program	static data global & static variables	dynamic data heap free storage (new/delete)	stack local variables (functions)
fixed size	fixed size	dynamic size	dynamic size

7.4. Pointers and arrays

In C++, an array name is analogue to a pointer to the first element of the array:

```cpp
char myName[4] = "Tim";
char * charPtr = NULL;
charPtr = myName;  // myName == &myName[0]
// note that this is invalid: myName = charPtr;

std::cout << "1st: " << *(charPtr) << '\n';
// → gives out 'T'
std::cout << "2nd: " << myName[1] << '\n';
// → gives out 'i'
std::cout << "3rd: " << *(charPtr+2) << '\n';
// → gives out 'm'
```

Memory

myName[0],..,myName[3]

38

| 0101 0100 | 0110 1001 | 0110 1101 | 0000 0000 |

'T'   'i'   'm'   0

myName=char*

| 0000 0000 | 0010 0110 |

7.4. Pointers and arrays

In C++, an array name is analogue to a pointer to the first element of the array:

```cpp
char myName[4] = "Tim";
char * charPtr = NULL;
charPtr = myName;  // myName == &myName[0]
// note that this is invalid: myName = charPtr;

std::cout << "1st: " << *(charPtr) << '\n';
// → gives out 'T'
std::cout << "2nd: " << myName[1] << '\n';
// → gives out 'i'
std::cout << "3rd: " << *(charPtr+2) << '\n';
// → gives out 'm'
```

Memory

myName[0],..,myName[3]

| 38 | → | 0101 0100 | 0110 1001 | 0110 1101 | 0000 0000 |

'T'   'i'   'm'   0

charPtr=char*

| 0000 0000 | 0000 0000 |

myName=char*

| 0000 0000 | 0010 0110 |

7.4. Pointers and arrays

In C++, an array name is analogue to a pointer to the first element of the array:

```cpp
char myName[4] = "Tim";
char * charPtr = NULL;
charPtr = myName;  // myName == &myName[0]
// note that this is invalid: myName = charPtr;

std::cout << "1st: " << *(charPtr) << '\n';
// → gives out 'T'
std::cout << "2nd: " << myName[1] << '\n';
// → gives out 'i'
std::cout << "3rd: " << *(charPtr+2) << '\n';
// → gives out 'm'
```

Memory

myName[0],..,myName[3]

0101 0100	0110 1001	0110 1101	0000 0000

'T'   'i'   'm'   0

38

charPtr=char*

0000 0000	0010 0110

myName=char*

0000 0000	0010 0110

7.4. Pointers and arrays

Arrays can be dynamically allocated at run time with pointers:

```cpp
// we receive a size variable here that we need an array around,
// but do not know how large it is at design time:
auto size = fileData.getSize();

// We CAN create an array of the required size:
GPSCoord * myRoute = new GPSCoord[size];  // a route is created as points
for (auto i = 0; i < size; i++) {  // note range-based for loop wouldn't work
  fileData.readNext();  // read data from file, set these as route points:
  myRoute[i].set( fileData[0], fileData[1] );
  myRoute[i].setElevation( fileData[2] );
}

delete[] myRoute;  // for dynamically created arrays, delete needs []
myRoute = NULL;
```

7.4. Pointers and arrays: Example 00 (difficulty level: 🌶️)

```cpp
/* Create an array for which the length is given at runtime through an argument
   of the executable. The main function in C++ can also have two parameters:
   argc: integer containing the count of arguments in argv
   argv: array of strings holding command-line arguments.
   argv[0] is the command itself, argv[1] is first argument */
#include <iostream>

int main(int argc, char * argv[]) { // executable's arguments are passed
  // if an argument given, assume it is a number and convert from string:
  if (argc > 1) {
    auto size = std::stoi(argv[1]);

    // add code to create an array of length size, and fill it with increasing
    // numbers from 1 till size, display these, and then delete the array


  }
}
```

7.5. References

In C++, a reference is like an alias, or second name, for a variable. A reference can only be initialized, but never reassigned to another variable.

```cpp
int myNumber = 7402312;
int & myPhone = myNumber;   // & in this declaration
// shows that this is a reference. From here on,
// myNumber and myPhone name the same variable.
int myOtherNumber = 2718354;
myPhone = myOtherNumber;
// → Now all three variable names myNumber, myPhone,
// and myOthernumber, have the same value: 2718354
// &myPhone = myOtherNumber; is invalid
```

Memory

myNumber,myPhone=int

0000 0000	0111 0000	1111 0011	0100 1000

myOtherNumber=int

0000 0000	0010 1001	0111 1010	1001 0010
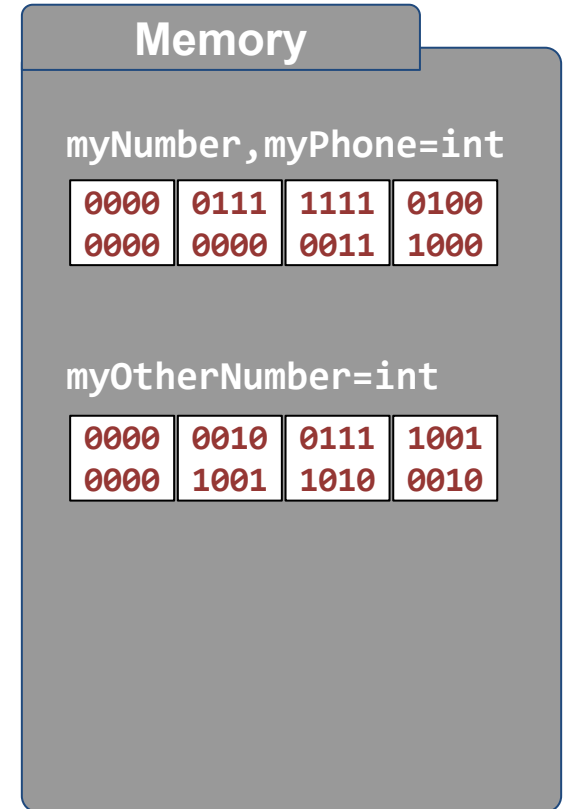
7.6. Call-by-Reference

Reminder: the swap function below is not going to work, because variables are **passed-by-value**

```cpp
#include <iostream>  // output to terminal
void swap(int x, int y){
  auto temp = 0;
  temp = x; x = y; y = temp;
}
int main() {
  auto a = 5, b = 10;
  swap(a, b);
  std::cout << a << ", " << b << '\n';
}
```

Memory

x | 10
y | 5
temp | 5
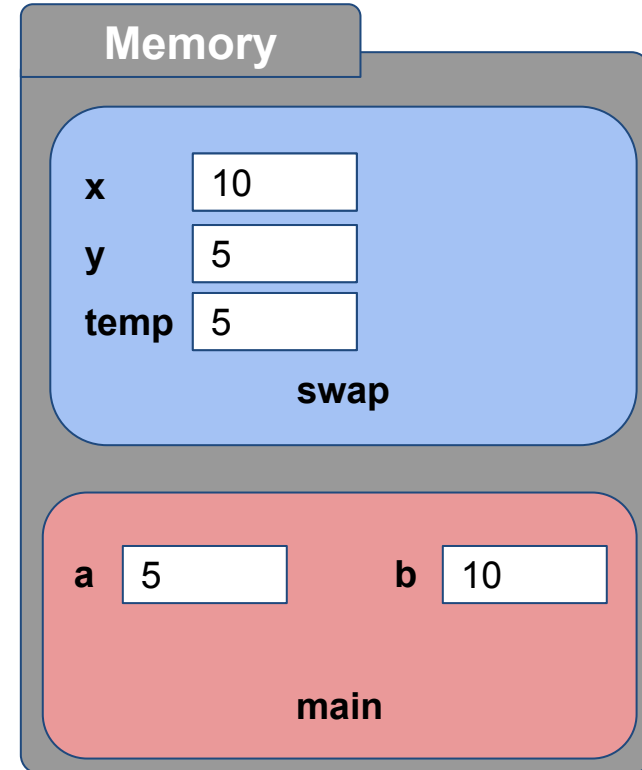
swap

a | 5   b | 10

main

7.6. Call-by-Reference

In C++, parameters can also be pointers. These are passed *by reference*, allowing swap to work:

```cpp
#include <iostream>  // output to terminal
void swap(int * x, int * y)    {
  auto temp = 0;
  temp = *x; *x = *y; *y = temp;
}
int main() {
  auto a = 5, b = 10;
  swap(&a, &b);
  std::cout << a << ", " << b << '\n';
}
```

Memory

*x

*y

temp 5
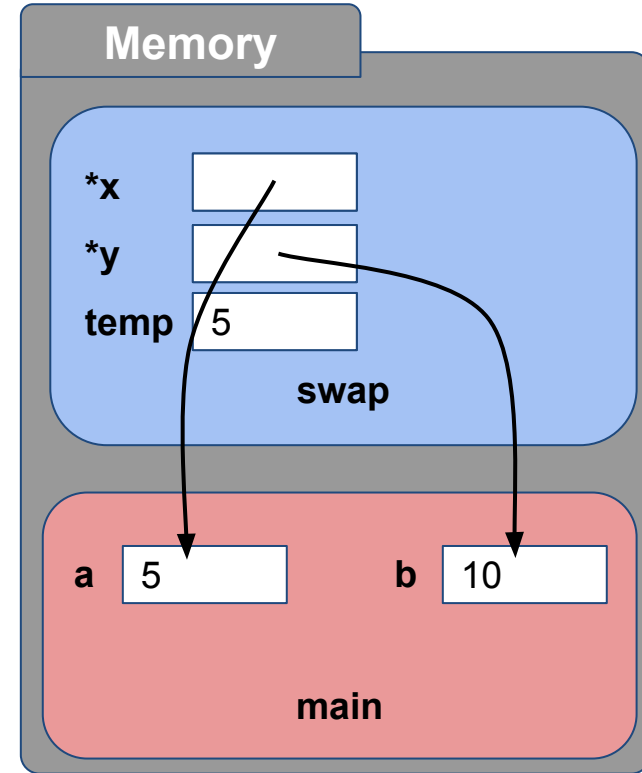
swap

a 5   b 10

main

7.6. Call-by-Reference

References have the same effect: These allow swap to work, too, and are safer and more elegant:

```cpp
#include <iostream>  // output to terminal
void swap(int & x, int & y)    {
  auto temp = 0;
  temp = x; x = y; y = temp;
}
int main() {
  auto a = 5, b = 10;
  swap(a, b);
  std::cout << a << ", " << b << '\n';
}
```

Memory

x

y

temp   5

swap

a   5   b   10

main
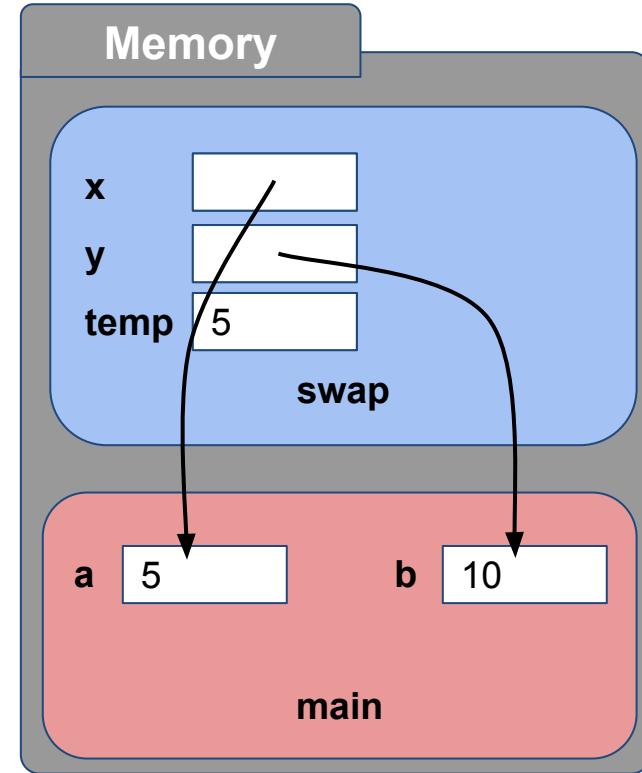
7.6. Call-by-Reference

Calling by reference avoids copying, which is often preferred. When variables do not change, they can be passed as **const** references.

This is generally a clearer signature for developers calling our function/method:

```cpp
#include <iostream>  // output to terminal
[[nodiscard("Please handle this function's return value.")]]
auto printOut(const std::string & x, const std::string & y) {
  return x + ',' + y + '\n';
}

int main() {
  std::string a = "5", b = "10";
  std::cout << printOut(a, b);
}
```

7.6. Call-by-Reference

Example 01 (difficulty level: 🌶️)

```cpp
/* Write a class, called Check, below to illustrate in the main function that
   the pointer to an object of class Check b is indeed pointing to the object of
   class Check a. */
#include <iostream>

// write the class Check here

int main() {
  Check a;  // a is an object of class Check
  Check * b = &a;  // assign address of a to pointer b to object of class Check
  if ( b->isThisMe( &a ) ) {
    std::cout << "&a is b \n";
  }
}
```

7.6. Call-by-Reference

Functions / methods create copies of parameters (pass by value). This also is the case for objects.

```cpp
#include <iostream>  // output to terminal
struct Pair { int x, y; };
void swap(Pair p)  {
  auto temp = 0;
  temp = p.x; p.x = p.y; p.y = temp;
}
int main()  {
  Pair p; p.x = 5; p.y = 10;
  swap(p);
  std::cout << p.x << ", " << p.y << '\n';
}
```

Memory

p | 5 | 10 |

temp | 5 |

swap

p | 5 | 10 |
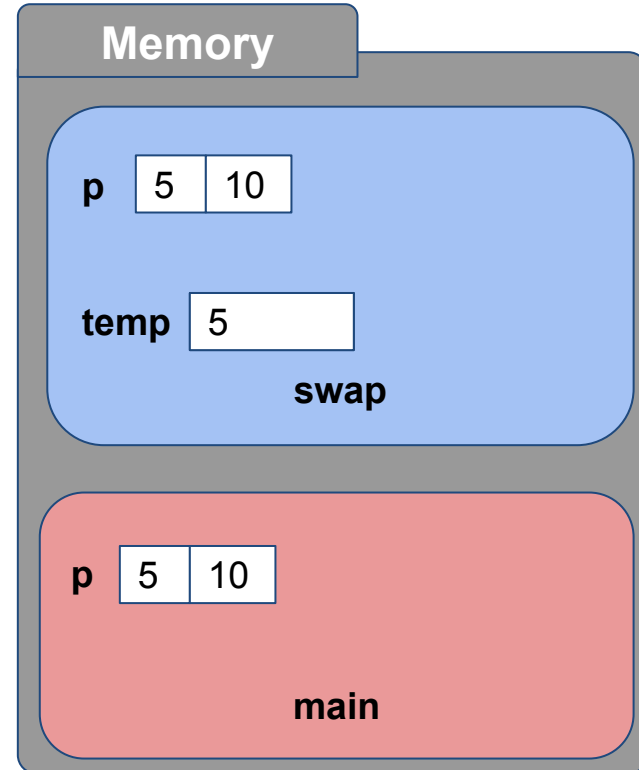
main

7.6. Call-by-Reference

Using a reference as function/method parameter allows the function to access the original object:

```cpp
#include <iostream>  // output to terminal
struct Pair { int x, y; };
void swap(Pair & p){
  auto temp = 0;
  temp = p.x; p.x = p.y; p.y = temp;
}
int main() {
  Pair p; p.x = 5; p.y = 10;
  swap(p);
  std::cout << p.x << ", " << p.y << '\n';
}
```

Memory

p

temp 5

swap

p 5 10
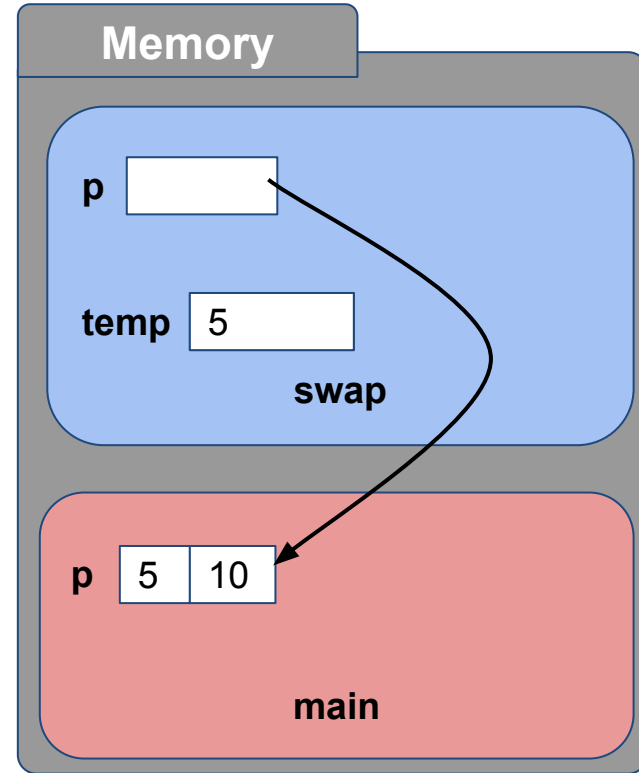
main

7.7. Copy Constructors

So when passing an object, it gets automatically copied.

How can we implement this copying of objects?

```cpp
void UTMCoord::from(GPSCoord coord) {
  // transforms from latitude-longitude to UTM coordinates
  // …
}


int main() {
  GPSCoord place(50.88385, 8.02096);
  UTMCoord place2;
  place2.from( place );  // ← place is here copied into a new object
                         //   and passed

}
```

7.7. Copy Constructors

How can we implement the copying of objects? ⇒ With **copy constructors**

```cpp
class GPSCoord {  // GPS coordinate class
 public:
  GPSCoord() {}  // default constructor
  GPSCoord(GPSCoord const & source);  // copy constructor
  void set(double lat, double lng);  // set latitude, longitude
  void setElevation(double elv);     // set elevation
  void print();  // output coordinates to console
 private:
  double lat, lng, elv;  // latitude, longitude, elevation
};
```

GPSCoord.h

```cpp
GPSCoord::GPSCoord(GPSCoord const & source) {
  lat = source.lat; lng = source.lng; elv = source.elv;
}
```

in GPSCoord.cpp

7.7. Copy Constructors

A copy constructor makes an object from another object of the same class, so in our example, we "clone" our GPSCoord object.

The copy constructor is implicitly called:

● when an object is **passed** to a function or method by value, or
● when a function or method **returns** an object

If a class does not implement a copy constructor, the C++ compiler will provide a default copy constructor, performing a ***member-wise copy*** (also known as ***shallow copy***)

So why do we ever have to implement a copy constructor ourselves?

7.7. Copy Constructors

An example of deep versus shallow copy:

```cpp
class GPSTrace {  // class for a GPS trace

 public:
  GPSTrace(uint16_t numPoints);
  ~GPSTrace();
  // add a new point to trace at position pos:
  void setPoint(GPSCoord newPoint, uint16_t pos);
  [[nodiscard]] int print();  // print trace, forces return handling

 private:
  GPSCoord *points;  // pointer to GPS coordinates
  uint16_t numPoints;
};
```

7.7. Copy Constructors

An example of deep versus shallow copy:

```cpp
GPSTrace::GPSTrace(uint16_t numpoints): numPoints(numpoints) {
  points = new GPSCoord[numPoints];
}

GPSTrace::~GPSTrace() {
  delete[] points; points = NULL; numPoints = 0;
}

void GPSTrace::setPoint(GPSCoord newPoint, uint16_t pos) {
  if (pos < numPoints) points[pos] = newPoint;
}

int GPSTrace::print() {  // output trace to console
  for (auto i = 0; i < numPoints; i++) points[i].print();
  return 0;
}
```

7.7. Copy Constructors

Example 02 (difficulty level: 🌶️🌶️)

```cpp
/** An exercise illustrating shallow and deep copy: Add to the code of GPSTrace
    the necessary functionality that allows copying a GPSTrace and illustrate this
    in the main function below. */
#include <iostream>      // terminal output

// Classes GPSCoord and GPSTrace come here

int main() {
  GPSTrace t(5);
  for (auto i = 0; i < 5; i++ ) {  // fill in the GPS points
    GPSCoord point;
    point.set( i*1.2, i*3.4 ); point.setElevation( i*5.6 );
    t.setPoint( point, i );
  }
  return t.print();
}
```

7.7. Copy Constructors

Example 02 (difficulty level: 🌶️🌶️)

The standard (shallow) copy of `p` (e.g., `pCopy`) will result in this:
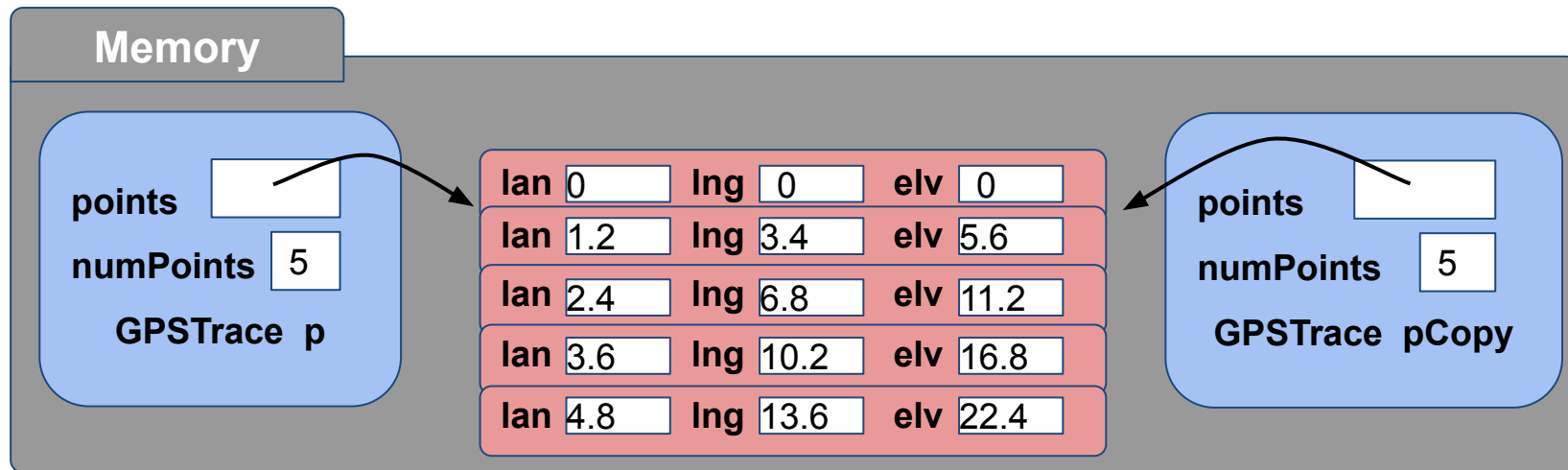
So the (deep) copy needs to be implemented in the copy constructor

7.8. `const` and `const` Pointers

Reminder: constants are defined with the **const** keyword, e.g.:

```
const int gpsTraceLength = 150;   // an integer constant
```

- A constant can only be initialized, a new value can not be assigned to a constant once it is defined.
- Usually this is done to protect the constant from being changed later on when this isn't planned.

7.8. `const` and `const` Pointers

`const` pointers can come in various forms, what matters is that everything on the left of the `const` keyword is constant. If `const` is on the full left, what is on its right is constant. `const` pointers need to be directly initialized:

```cpp
int myInteger = 71;
const int constInt = 17;
int const *pointToConstInt = &constInt;   // pointer to const int
int * const constPointToInt = &myInteger;  // const pointer to int
int const * const cPointToCInt = &constInt;  // const pointer to const int
const int * pointToConstInt2 = &constInt;  // pointer to const int
```

7.8. `const` and `const` Pointers

`const` pointers protect pointers from being changed later in the code.

This means that changes such as: `*pointToConstInt = myInteger;`, `constPointToInt = &myInteger;`, `*cPointToCInt = myInteger;`, and `cPointToCInt = &myInteger;` will all result in an error.

For example:
The `this` pointer in a class' methods is a `const` pointer to an object of the class, so cannot be changed to point to anywhere else but the current object.

7.8. **const** and **const** Pointers

const member functions or methods (aka *const qualified*):

```
int GPSTrace::print() const;  // output trace to console
```

const in the method's declaration *and* definition tells the compiler that the method should not modify the object (i.e., change the attributes)

- The compiler enforces this, and reports an error once the method's code tries to change its attributes
- Using const methods whenever possible will add guarantee that it will be used properly in the future

7.8. **const** and **const** Pointers

Example 03 (difficulty level: 🌶️🌶️🌶️):

```cpp
/** Print a mouse in the console, using a const pointer to avoid changes */
#include <iostream>      // terminal output
[[nodiscard]] auto * getBitmapAddress() {
    static auto bitmap[] = "(^._.^)~\n";  // "bitmap" created in static memory
    return bitmap;  // return pointer to first element
}

int main() {
  // using a pointer to bitmap, and incrementing it, is possible:
  auto * mousePointer = getBitmapAddress();
  while ( *mousePointer != 0 )  std::cout << *(mousePointer++);
  // Here mousePointer has changed, it's hard to get the original pointer.
  // Modify the above by protecting the pointer with const, and looping twice.
}
```

7.9. Passing functions to functions: Passing pointer to function

In C, functions can be passed as a parameter, which will be as a pointer to the function and is just the function's name:

```cpp
#include <iostream>      // terminal output

int addTwo(int x) { return x + 2; }     // functions we can pass in callFunct
int timesFour(int x) { return x * 4; }  // since they match the signature

// callFunct takes a pointer to a function:
int callFunct(int x, int (*funct)(int) ) { return funct(x); /* = (*funct)(x) */ }

int main() {
  std::cout << "addTwo(149) = " << callFunct(149, addTwo) << '\n';
  std::cout << "timesFour(4) = " << callFunct(4, timesFour) << '\n';
}
```

7.9. Passing functions to functions: The `std::function`

In C++ 11 and onward, `std::function` can be used from `<functional>` (using templates, see later):

```cpp
#include <iostream>      // terminal output
#include <functional>      // use std::function to pass functions as parameter

int addTwo(int x) { return x + 2; }     // functions we can pass in callFunct
int timesFour(int x) { return x * 4; }  // since they match the signature

// callFunction takes a pointer to a function:
int callFunct(int x, std::function<int(int)> func ) { return func(x); }

int main() {
  std::cout << "addTwo(149) = " << callFunct(149, addTwo) << '\n';
  std::cout << "timesFour(4) = " << callFunct(4, timesFour) << '\n';
}
```

7.9. Passing functions to functions: Passing pointer to function

Example04 (difficulty level: 🌶️🌶️🌶️):

```cpp
/** Define the class' methods below so that the main function makes sense */
#include <iostream>      // terminal output
#include <functional>     // use std::function to pass functions as parameter
class NumberSequence {  // class for sequence of whole, positive numbers

 public:
  NumberSequence(uint16_t length = 10);
  // apply the function func() to all numbers:
  void forEach(std::function<uint16_t(uint16_t)> func);
  void print() const;  // print all numbers to console

 private:
  const uint16_t length;  // length of number sequence
  uint16_t *seq;  // the numbers are stored as a dynamic array
};
```

7.9. Passing functions to functions: Passing pointer to function

Example04 (difficulty level: 🌶️🌶️🌶️):

```cpp
// define all NumberSequence methods here
//

uint16_t times2(uint16_t n) { return n*2; }

int main() {
  NumberSequence s;
  s.print();
  s.forEach( &times2 );  // apply the function times2 to all numbers
  s.print();
}
```

7.9. Passing functions to functions: Passing pointer to function

On the pointers behind functions and arrays:

```cpp
#include <iostream>
int fun(int i) { return i+7; }  // a function
int main() {
  int a[3] = {1,2,3};  // an array

  // conversion to function pointers defaults to a bool:
  std::cout << "fun:        " << fun << '\n';  // will result in warning
  std::cout << "&fun:       " << &fun << '\n';
  // Converting to a void pointer works:
  std::cout << "(void*)fun: " << (void*)fun << '\n';
  std::cout << "(void*)&fun:" << (void*)&fun << '\n';
  // The address of an array can be gotten by:
  std::cout << "a:      " <<  a << '\n';
  std::cout << "&a:     " << &a << '\n';
  std::cout << "&a[0]: " << &a[0] << '\n';
}
```

7.10. Smart Pointers in C++

Smart pointers are a wrapper class over pointers, to avoid memory leaks, wild (never initialized), or dangling (pointing to deleted memory) pointers.

They destroy themselves when they go out of scope, and can efficiently manage memory through extra functionality.

They are a part of the `<memory>` module (in `<iostream>`) and implement:
- ~~`auto_ptr`~~: deprecated after C++11.
- `unique_ptr:` an exclusive pointer that cannot be copied (just moved) and cleans up after itself
- `shared_ptr:` a pointer that can be shared: Multiple pointers can point to the same object, and this is managed (and counted).
- `weak_ptr`: beyond the scope of this course

7.10. Smart Pointers in C++

```cpp
#include <iostream>    // terminal output and smart pointers
struct Coordinate {  // class for an x,y coordinate
  int x = 10, y = 20;
  void print() {
    std::cout << x << ',' << y << '\n';
  }
};

int main() {
  std::unique_ptr<Coordinate> p1(new Coordinate);
  std::unique_ptr<int[]> p2(new int[15]);  // int[], so delete[] is used
  p1->print();
  std::cout << p2.get() << '\n';  // display pointer p2's address
  // this would give a compiler error:
  //   unique_ptr<Coordinate> p3 = p1;
  // no need to delete p1 or p2 here, that is done for us
}
```

8.1. Inheritance

8.2. The `protected` keyword

8.3. (Delegate) constructors and destructors

8.4. `mutable`, `using`, `friend`, and `delete`

8.5. Polymorphism

8.1. Inheritance   https://isocpp.org/wiki/faq/basics-of-inheritance

- Generalization: Relationship between a more general class (base class, or superclass) and a more specialized class (subclass)
  - the specialized class is consistent with the base class, but contains additional attributes, operations and/or associations
  - an object of the subclass can be used wherever an object of the super class is permitted

- Generalization is not about just summarizing common properties and behaviors, but about generalizing in the literal sense:
  Every object of the subclass is an object of the superclass

8.1. Inheritance

```cpp
class Person {
 private:
  std::string name;
  std::string address;
  int birthYear;
 public:
  void printBadge();  // prints name
  int getAge();  // returns age in years
};
```

```cpp
int main() {
  Staff newStaff;
  Trainee newTrainee;
  // newStaff and newTrainee objects
  // can access Person public method
  newStaff.printBadge();
  newTrainee.getAge();

}
```

```cpp
class Staff: public Person {
 private:
  double salary;
 public:
  void setSalary(double s);
};
```

```cpp
class Trainee: public Person {
 private:
  int startDay;
 public:
  int daysInTraining();
};
```

8.2. The `protected` keyword

- Private members (attributes or methods) are only accessible within the class that has defined them

- Public members are accessible from anywhere

- Protected members are accessible in the class that defines them, and in classes that inherit from that class

8.2. The `protected` keyword: Inaccessible from outside the class..

```cpp
class Person {
 private:
  std::string address;
   int birthYear;
  protected:
  std::string name;
  public:
  void printBadge();  // prints name
  int getAge();  // returns age in years
};
```

```cpp
int main() {
  Staff newStaff;
  // newStaff object cannot access
  // Person protected attributes
  // or methods from outside class:
  // std::cout << newStaff.name;
  //   -> ERROR

}
```

```cpp
class Staff: public Person {
 private:
  double salary;
 public:
  void setSalary(double s);
};
```

```cpp
class Trainee: public Person {
 private:
   int startDay;
 public:
   int daysInTraining();
};
```

8.2. The **protected** keyword: .. but accessible from child classes

```cpp
class Person {
 private:
  std::string address;
  int birthYear;
 protected:
  std::string name;
 public:
  void printBadge();  // prints name
  int getAge();  // returns age in years
};
```

```cpp
int Trainee::daysInTraining() {
  int returnVal;
  // Trainee cannot access Person's
  // private attributes:
  // returnVal = birthYear -> ERROR
  // but Trainee can access Person's
  // protected attributes in class:
  std::cout << name << std::endl;
  return (today()-startDay);
}
```

```cpp
class Staff: public Person {
 private:
  double salary;
 public:
  void setSalary(double s);
};
```

```cpp
class Trainee: public Person {
 private:
  int startDay;
 public:
  int daysInTraining();
};
```

8.3. (Delegate) constructors and destructors

Remember the special syntax in constructors, to initialize attributes:

```cpp
class Example {
 public:
  Example(int & aVar);
 private:
  const int aConst;   // a constant
  int & aRef;         // a reference
};
// This would lead to errors:
// Example::Example(int &aVar) {
//    aConst = 47; aRef = aVar;
// }
// But this works:
Example::Example(int & aVar)
  : aConst(47), aRef(aVar) {
  // here can follow more code
}
```

Passing arguments to the base class constructor is possible with:

```cpp
class BaseClass {
 public:
  BaseClass(int var) : var(var) {}
 private:
  int var;
};

class SubClass : public BaseClass {
 public:
  SubClass(bool myBool)
    : BaseClass(7), myBool(myBool) {}
 private:
  bool myBool;
};
```

8.3. (Delegate) constructors and destructors

Passing arguments to a base class constructor is possible with:

```cpp
class BaseClass {
 public:
  BaseClass(int var) : var(var) {}
 private:
  int var;
};

class SubClass : public BaseClass {
 public:
  SubClass(bool myBool)
   : BaseClass(7), myBool(myBool) {}
 private:
  bool myBool;
};
```

Similarly, *delegate constructors* call others from the same class to reduce repetitive code (from C++11 onward):

```cpp
class MyClass {

 public:
  MyClass(int a1, bool b1) : a(a1), b(b1) {
    // lots of initialization work
  }
  MyClass(int a1) : MyClass(a1, true) {}
  MyClass(bool b1) : MyClass(10, b1) {}

 private:
  int  a;
  bool b;
  // ...
};
```

8.3. (Delegate) constructors and destructors
Example 00 (difficulty level: 🌶️)

```cpp
#include <iostream>
class Book {  // a Book object always has a title and a price.
 // change this class so that the title cannot be changed, and the price can be changed by Magazine:
  Book(std::string name, double val) : title(name), price(val) {}
  void show() { std::cout << title << " - " << price << "\n"; }
  std::string title;
  double price;
};

class Magazine : public Book {  // a Magazine object uses the Book's constructor, and can apply a discount
  // change this class so that an object is created solely through Book's constructor
  void discount(double percent);
};
// implement Magazine's discount method here

int main() {
  Magazine mag(std::string("C++ Monthly"), 10.0);
  mag.show(); mag.discount(25.0);  // this should show 10 in the console, then we apply a 25 % discount
  mag.show();  // this should now show 7.5
}
```
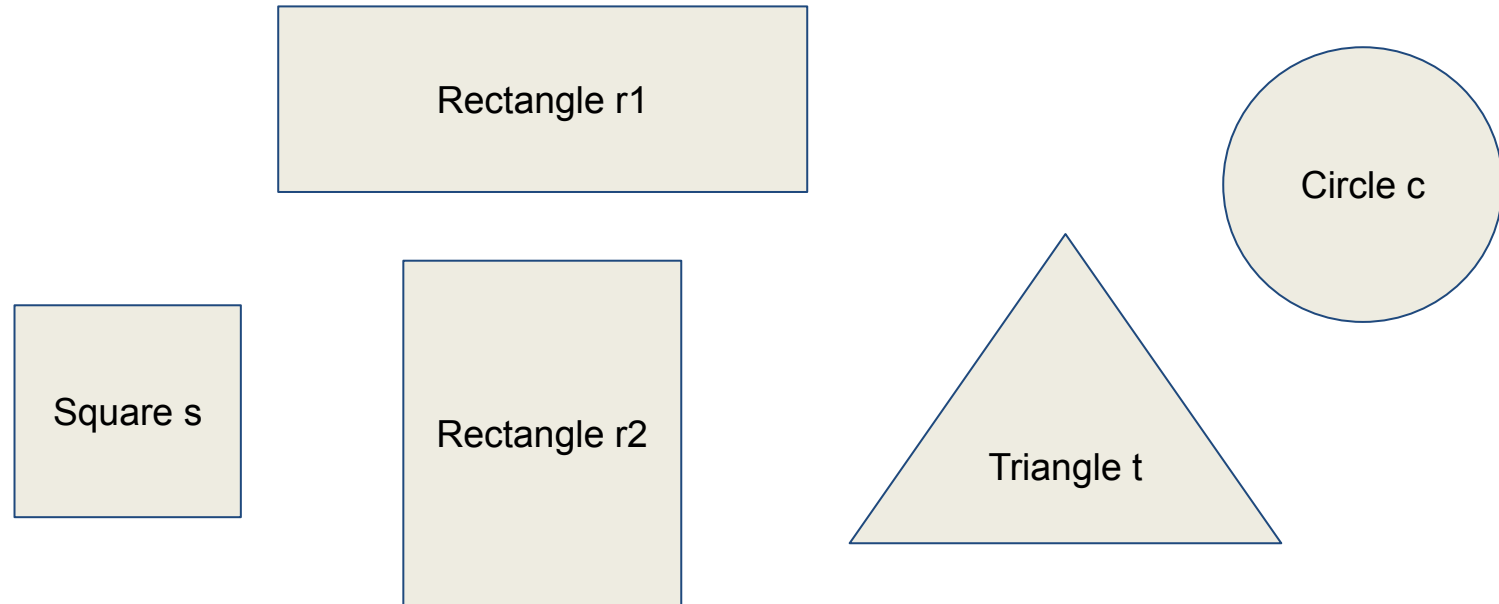
8.3. (Delegate) constructors and destructors

Example 01 (difficulty level: 🌶)

Creating 2D graphics elements such as the ones below as classes' objects

8.3. (Delegate) constructors and destructors Example 01

```cpp
class Element {  // class representing graphic element
 public:
  Element(double x, double y) : x(x), y(y) {}
 private:
  double x, y;  // position of graphic element
};

class Rectangle : public Element {  // class representing a rectangle
 public:
  Rectangle(double x, double y, double a, double b) : Element(x, y), a(a), b(b) {}
 private:
  double a, b;  // width and height of rectangle
};

class Square : public Rectangle {  // class representing a square
 public:
  Square(double x, double y, double a) : Rectangle(x, y, a, a) {}
};
```

🏠 Assignment: Add a method to Square that prints out its location: What needs changing?

8.4. **mutable**, **using**, **friend**, and **delete**

Any **mutable** data members of **const** class instances can be modified. This is useful if most of the class' members should be constant, but a few need to be modified.

```cpp
#include <iostream>

class A {  // class A
 public:
  int x = 4;   // public attributes, default initialized (since C++11)
  mutable int y = 3;
};

int main() {
  const A a;  // constant object of class A
  a.y = 7;    // this works, a.x would result in compile error
  std::cout << a.y << '\n';
}
```

8.4. `mutable`, `using`, `friend`, and `delete`

The `using` keyword can be used to change the inheritance properties of class attributes or methods:

```cpp
class A {  // class A
 protected:
  int x = 4;   // protected attribute
};

class B : public A {  // class A
 public:
  using A::x;   // inherits x and exposes it as a public attribute
};

int main() {
  B b;
  b.x = 7;     // this works, b.x is public (even though A.x is protected)
}
```

8.4. `mutable`, `using`, `friend`, and `delete`

The `friend` keyword allows a class to access the private and protected attributes and methods of the class that is declared as a friend.

⚠️ A `friend` relation is **not**:

symmetric: Class A as a friend of B does not imply class B being a friend of A

transitive: A is a friend of B and B is a friend of C does not imply A is a friend of C

inherited: Class Base as a friend of class X does not imply subclass Derived is a friend of class X; Class X as a friend of class Base does not imply class X is a friend of subclass Derived

8.4. `mutable`, `using`, `friend`, and `delete`

```cpp
#include <iostream>
class A {  // class A declares that B is a friend
 private:
  friend class B;
  int x = 4;  // private attribute
};

class B {  // class B is a friend of A
 public:
  B() { A a; y = a.x;}        // and thus can access
  int f(A a) { return a.x; }  // A's private attribute x
 private:
  int y;
};

int main() {
  A a; B b;
  std::cout << b.f(a) << '\n';
}
```

8.4. `mutable`, `using`, `friend`, and `delete`

A **friend** *method* can access the private and protected members of a class if it is declared a friend of that class.

```cpp
#include <iostream>

class A {  // class A declares funct as a friend method
 private:
   int x = 4;  // private attribute
   friend int funct(A a);
};

int funct(A a) { return a.x; }  // funct is not a class method

int main() {
   A a;
   std::cout << funct(a) << '\n';
}
```

8.4. **mutable**, **using**, **friend**, and **delete**

Example 02 (difficulty level: 🌶️)

```cpp
#include <iostream>

class Rectangle {  // class Rectangle has width and height as attributes and area() as a method
 public:
  Rectangle() {}  // default constructor, allows to define width and height later
  Rectangle(int x, int y): width(x), height(y) {}  // constructor that sets attributes
  int area() { return width*height; };
  // declare the friend method "enlarge()" here, with a rectangle as parameter, returning a rectangle
 private:
  int width, height;  // width and height are private, so not accessible from outside the class
};

/* define the friend method here, so that it creates and returns a copy of the rectangle that
   has twice the width and height. The friend method has access to the private attributes. */

int main() {
  Rectangle rectangle1, rectangle2(3,4);  // rectangle1 will obtain twice the width and height
  rectangle1 = enlarge(rectange2);        // of rectangle2 through the enlarge method
  std::cout << rectangle1.area() << '\n';  // this should return "48" (6 times 8)
}
```

8.4. `mutable`, `using`, `friend`, and `delete`

The **delete** keyword marks a class method as deleted. Calling that method (explicitly or implicitly) will result in a compiler error. The **delete** keyword prevents implicitly generating default copy constructors or assignments.

```cpp
class Element {  // class representing graphic element
 public:
  Element(double x, double y) : x(x), y(y) {}  // default constructor is deleted
  Element(Element const & e) = delete;  // copying an object gives a compiler error
  void setX(double x) { this->x = x;}
  void setX(float x) = delete;  // calling with a float results in a compiler error
 private:
  double x, y;  // position of graphic element
};

int main() {
  Element e(2.0, 3.0);
  e.setX(5.0);  // works, but: e.setX(5.0f) would fail (due to delete above)
}
```

8.4. `mutable`, `using`, `friend`, and `delete`: Maze Game v.5.00

Class Player and Maze were changed, so we can load a map and add player 2:

```cpp
/* Fifth draft of Maze Game: we now use class "Player" and use a file */   mazeGame.cpp
#include "Maze.h"   // class that holds everything related to the maze
#include "Player.h"   // class that holds everything related to the player
int main() {
  auto c = ' ';        // used for user key input
  std::string mapFile("maze.csv");
  Maze maze(mapFile);    // initialize a maze object from this file
  Player player1(10, 5);
  Player player2(20, 7);
  while ( c != 'q' ) {  // as long as the user doesn't press q ..
    maze.draw();     // draw maze
    player1.draw('@', 3);  // draw player 1 as a '@' with color pair 3
    player2.draw('X', 3);  // draw player 2 as a 'X' with color pair 3
    c = getch();           // capture the user's pressed key
    switch (c) {
      // …
    }
```

8.4. `mutable`, `using`, `friend`, and `delete`: Maze Game v.5.00

Modify the class Player and add a child class Enemy, so that moving the players is now part of the draw class, and the enemy can move toward the player:

```cpp
/* Fifth draft of Maze Game: we now use class "Player" and use a file */   mazeGame.cpp
#include "Maze.h"  // class that holds everything related to the maze
#include "Player.h"  // class that holds everything related to the player
int main() {
  auto c = ' ';         // used for user key input
  std::string mapFile("maze.csv");
  Maze maze(mapFile);    // initialize a maze object from this file
  Player player(10, 5, 'w','s','a','d');  // player at (10,5) and moves with wasd
  EnemyPlayer enemy(20, 7);  // enemy player starts at (20,7)
  while ( c != 'q' ) {  // as long as the user doesn't press q ..
    maze.draw();      // draw maze
    player.draw('@', 3, c);       // draw player 1 as a '@' with color pair 3
    enemy.draw('X', 3, player); // draw enemy player as a 'X' with color pair 3
    c = getch();                 // capture the user's pressed key
  }
```

8.5. Polymorphism

C++ provides a way to create a base object with methods that through overriding change their behavior. At run-time, objects of the base class behave as objects of a derived class

If **Sub** is a subclass of **Super**, then the following assignments are allowed:

```
Sub sub;
Super * superPtr = &sub;   // a parent class pointer is allowed to
Super & superRef = sub;    // point to a child class' object
```

A base class pointer or reference can also point or refer to a subclass' object.

The other way round is not allowed.

8.5. Polymorphism

For example: **Dog** and **Fish** are classes that inherit from **Animal**. We want any objects from these classes to have a **print()** method, which displays the values of the classes' attributes. Then an **Animal** object pointer could be used to flexibly point to a **Dog** or **Fish** object and access the right **print()** method:

```cpp
Animal * animal;
animal = new Dog("Scooby");
animal->print();  // prints out: I am Scooby. Bark!
animal = new Fish("Salmon");
animal->print();  // prints out: I'm Salmon (fish)
```

For polymorphism to work in C++, the base class must declare the methods in question as being **virtual**

8.5. Polymorphism: Example (1/2)

polyDemo.cpp

```cpp
#include <iostream>
#include <cstdlib>

class Animal {  // Animal class stores the species and prints this in print()
 protected:
  std::string _species;
 public:
  Animal(std::string species) { _species = species; }
  virtual void print() { std::cout << "I'm" +  _species << '\n'; }
};

class Dog : public Animal {  // Dogs inherit species from Animal and have a name
 protected:
  std::string _name;
 public:
  Dog(std::string name) : Animal("dog"), _name(name) {}
  void print() { std::cout << "I am" << _name << "Bark.\n"; }
};
```

8.5. Polymorphism: Example (2/2)

```cpp
                                                    polyDemo.cpp
class Fish : public Animal {  // Fishes have species and subspecies
 protected:
  std::string _subspecies;
 public:
  Fish(std::string subspecies) : Animal("fish"), _subspecies(subspecies) {}
  void print() { std::cout << "I'm" << _subspecies << "(fish)\n"; }
};


int main() {
  Animal * animals[4] = { new Dog("Snowy"), new Fish("Salmon"),
                          new Dog("Scooby"), new Animal("Some animal") };
  for (int i=0; i<15; i++) {
    Animal * a = animals[rand() % 4];  // a is a polymorph variable: its
    a->print();                        // print's behavior depends on the
  }                                    // object that a points to

}
```

8.5. Polymorphism: Example

- Note that animals is an array of pointers to the base class (Animal)

- Yet, we can let the pointers point to a subclass of Animal (Dog, Fish, …)

- And when we call print() from Animal pointer a, the right method executes

```
Animal * animals[4];
animals[0] = new Dog("Snowy");
animals[1] = new Fish("Salmon");
...
  Animal * a = animals[rand()%4];
  a->print();
```

```
~\> g++ polyDemo.cpp -o polymorph_demo
~\> ./polymorph_demo
I'm Salmon (fish)
I am Scooby. Bark!
I'm Salmon (fish)
I am Scooby. Bark!
I am Snowy. Bark!
I'm some animal
I am Scooby. Bark!
I'm Salmon (fish)
I am Snowy. Bark!
I am Scooby. Bark!
I am Snowy. Bark!
I am Scooby. Bark!
I'm some animal
I am Scooby. Bark!
I'm Salmon (fish)
~\>
```

8.5. Polymorphism: `virtual` and late binding

Connecting the function call to the function body is called ***Binding***

Early / static / compile-time binding is done by the compiler through object type, letting the program jump to the function's address

Late / dynamic / run-time binding uses the object type while the executable is running and then matches the function call with the correct function definition: The program has to read the address held in the pointer and then jump to that address. This extra jump makes it less efficient.

C++ achieves late binding by declaring a virtual function

8.5. Polymorphism: **virtual** and late binding

C++ achieves late binding by declaring a virtual function in the base class:

```cpp
#include <iostream>
class A {  // class A has a virtual function funct:
 public:
  virtual void funct() { std::cout << "A \n"; };
};
class B : public A {  // class B inherits from A and overrides funct:
 public:
  virtual void funct() { std::cout << "B \n"; };  // virtual here is optional
};

void g( A & a ) { a.funct(); }  // g accepts A and B as parameter

int main() {
  A a; B b;
  g(a); g(b);  // outputs first "A", and then "B"
}
```

8.5. Polymorphism: **`virtual`** and **`override`**

The **`override`** keyword (from C++11) ensures that the method is virtual and is overriding a virtual function from a base class:

```cpp
#include <iostream>
class A {  // class A has a virtual function funct:
 public:
  virtual void funct() { std::cout << "A \n"; };
};
class B : public A {  // class B inherits from A and overrides funct:
 public:
  void funct() override { std::cout << "B \n"; };
};

void g( A & a ) { a.funct(); }  // g accepts A and B as parameter

int main() {
  B b; g(b);  // outputs "B"
}
```

8.5. Polymorphism: `final`

The `final` keyword (from C++11) prevents inheriting from classes or overriding methods in derived classes

```cpp
class A final {  // class A cannot be extended
  // …
};
class B : public A {  // leads to compile error: A is "final"
  // …
};

class C {  // class C contains a virtual final method:
 public:
  virtual void funct() final;
};
class D : public C {
 public:
  void funct();  // leads to compile error: method funct is "final"
};
```

9.1. Assertions and debuggers

9.2. **try**, **throw**, **catch**

9.3. Function try blocks

9.4. **noexcept**

9.5. **std::nested_exception** and **std::throw_with_nested**

9.1. Assertions and debuggers

A way to ensure that a condition always should hold at some stage in the code:
If the expression supplied to **assert()** is false (0), the program *aborts* at the statement with an error.

```cpp
#include <iostream>  // use of std::cout, std::cin        example00.cpp
#include <cstdlib>   // std::rand()
#include <ctime>     // std::time()
#include <cassert>   // assert()
int main() {
  std::srand( std::time(nullptr) );  // time seeds random generator (nullptr)
  double myValue = ( std::rand() % 4 ) - 2;  // gets a random value
  assert(myValue != 0);  // since we'll divide by myValue, it shouldn't be zero
  myValue = 5 / myValue;  // integer division by 0 leads to undefined behavior
  std::cout << myValue << '\n';
}
```

9.1. Assertions and debuggers

Assert is a macro and depends on another macro, NDEBUG: If it is defined as a macro name (i.e., `#define NDEBUG`) before <cassert> is included, then assert will be disabled.

The program's state at particular points (e.g., after an assertion fails) can be checked in a ***debugger*** to allow watching the state of a running program. Examples: stop execution at a given code line (breakpoint), examine call stack, print / modify contents of variables, print type definitions, execute line-by-line
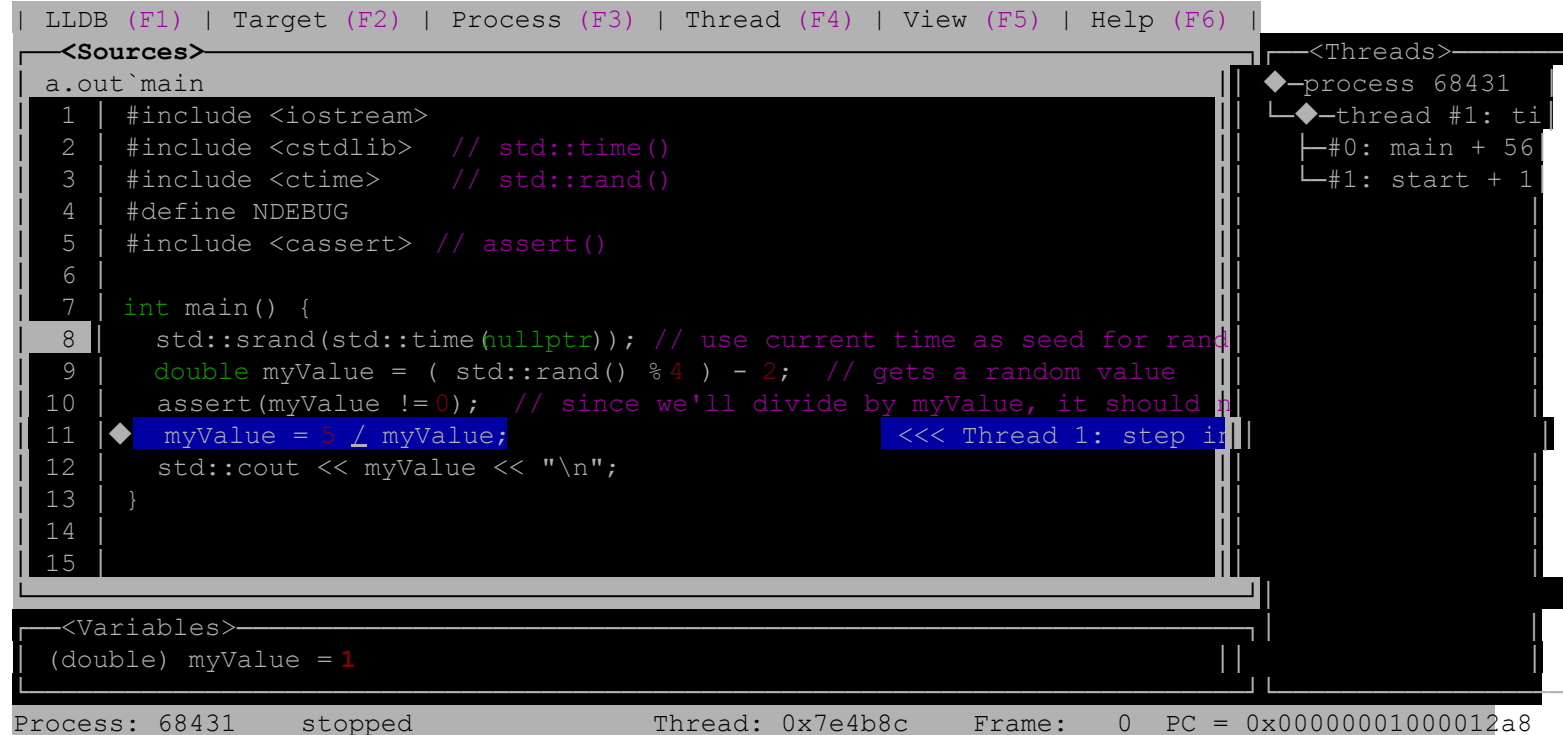
examples: ddd or gdbgui , both use gdb , or lldb → try on the previous code:

```
> g++ -g example00.cpp
> lldb a.out
```

9.1. Assertions and debuggers -- lldb example

```
> lldb a.out
bash-5.1$ lldb a.out
(lldb) target create "a.out"
Current executable set to '/Users/kvl/sciebo/UbiComp/Teaching/AdvancedCPP_43UCO1118V/a.out'
(x86_64).
(lldb) b main
Breakpoint 1: where = a.out`main + 15 at example00.cpp:8:14, address = 0x000000010000127f
(lldb) run
Process 68431 launched: '/Users/kvl/sciebo/UbiComp/Teaching/AdvancedCPP_43UCO1118V/a.out'
(x86_64)
Process 68431 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x000000010000127f a.out`main at example00.cpp:8:14
   5        #include <cassert>  // assert()
   6
   7        int main() {
-> 8          std::srand(std::time(nullptr)); // use current time as seed for random generator
   9          double myValue = ( std::rand() % 4 ) - 2;  // gets a random value
   10         assert(myValue != 0);  // since we'll divide by myValue, it should not be zero
   11         myValue = 5 / myValue;  // integer division by 0 leads to undefined behavior
Target 0: (a.out) stopped.
(lldb) gui
```

9.1. Assertions and debuggers -- lldb example

```
| LLDB (F1) | Target (F2) | Process (F3) | Thread (F4) | View (F5) | Help (F6) |
┌─<Sources>──────────────────────────────────────────┐   ┌─<Threads>──────────
| a.out`main                                          |   |◆─process 68431
|   1 | #include <iostream>                            |   └◆─thread #1: ti
|   2 | #include <cstdlib>   // std::time()            |     ├─#0: main + 56
|   3 | #include <ctime>     // std::rand()            |     └─#1: start + 1
|   4 | #define NDEBUG                                 |
|   5 | #include <cassert> // assert()                 |
|   6 |                                                |
|   7 | int main() {                                   |
|   8 |   std::srand(std::time(nullptr)); // use current time as seed for rand
|   9 |   double myValue = ( std::rand() % 4 ) - 2;  // gets a random value
|  10 |   assert(myValue !=0);  // since we'll divide by myValue, it should n
|  11 |◆  myValue = 5 / myValue;                     <<< Thread 1: step i
|  12 |   std::cout << myValue << "\n";
|  13 | }
|  14 |
|  15 |
└────────────────────────────────────────────────────┘
┌─<Variables>──────────────────────────
| (double) myValue = 1
Process: 68431    stopped          Thread: 0x7e4b8c    Frame:   0  PC = 0x00000001000012a8
```

9.1. Assertions and debuggers

Assertions versus exceptions

Both detect run-time errors in a program, *but*:

- Assert() aborts the program, for the developer to fix their code
- Exceptions allow the program to recover and continue the execution from the first matching catch
  - Examples are any areas where variables obtain values outside the developer's control (e.g., others supply your code with file names which do not exist, or array sizes that do not fit in memory)
  - When no exception matches, the program will still abort

9.2. `try`, `throw`, `catch`

- When an error occurs, functions or methods may *throw* an exception, to be handled later when *catch*ing the exception.
- If an exception is *throw*n in the *try*-block, the *try*-block is exited and the associated *catch*-block is executed. `catch (...)` will catch the remaining thrown exceptions.
- Exceptions that go uncaught will cause the program to halt.

```cpp
try {
  throw 0.07f;  // throw an exception of type float
}
catch (float f) {  // float is thrown
  std::cout << "Exception: " << f << '\n';
}
catch (...) { std::cout << "Exception\n"; }  // default catch
```

9.2. `try`, `throw`, `catch`

throw supplies an instance of an exception class. This can be a built-in type, but more commonly is a class derived from the **std::exception** class:

```cpp
#include <iostream>  // std::cout, std::runtime_error, std::exception, std::cerr

int divBy(int a, int b) {
  if (b == 0)
    throw std::runtime_error("Divided by 0.");  // exception type runtime_error
  return a / b;
}

int main() {
  try { divBy(7, 0);  // this function throws an exception when b == 0
  }
  catch (const std::exception& e) {
    std::cerr << "Exception handled: " << e.what() << '\n';
  }
}
```

9.2. `try`, `throw`, `catch`

Throwing a *custom exception* requires a custom exception class, which inherits from **std::exception** and overrides its **what** method to return an error message:

```cpp
#include <iostream>  // std::cout, std::runtime_error, std::exception, std::ce   example01.cpp

class MyException : public std::exception {
 public:
  MyException(const char * msg) : message(msg) {}   // Constructor sets exception message
  const char * what() { return message.c_str(); }   // Override what() to return own message
 private:
  std::string message;
};


int main() {
  try { throw MyException("Oops, my bad."); }   // create and throw object of MyException
  catch (MyException& e) { std::cerr << "Exception handled: " << e.what() << "\n"; }
}
```

9.2. `try`, `throw`, `catch`

The **std::exception** class has many subclasses for specific exceptions:

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error (since C++11)
- bad_typeid
- bad_cast
  - bad_any_cast (since C++17)
- bad_optional_access (since C++17)
- bad_expected_access (since C++23)
- bad_weak_ptr (since C++11)
- bad_function_call (since C++11)
- bad_alloc
  - bad_array_new_length (since C++11)

- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error (since C++11)
  - system_error (since C++11)
  - ios_base::failure (since C++11)
  - filesystem::filesystem_error (since C++17)
  - tx_exception (TM TS)
  - nonexistent_local_time (since C++20)
  - ambiguous_local_time (since C++20)
  - format_error (since C++20)
- bad_exception
- ios_base::failure (until C++11)
- bad_variant_access (since C++17)

9.3. Function try blocks

Function try blocks build an exception handler around a method or function's body,

instead of a block of code inside the function body.

```cpp
#include <iostream>  // std::cout, std::runtime_error, std::exception, std::cerr
class Superclass {
 public:
  Superclass(int x) : x(x) { if (x < 0) throw 1; }  // an exception can be thrown here
  int x;
};
class Subclass : public Superclass {
 public:         // what if we want to catch Superclass's constructor thrown exception here? ...
  Subclass(int x) : Superclass(x) {}
};
int main() {   // ... instead of here?
  try { Subclass sub(-5); } catch (int) { std::cout << "Oops, my bad.\n"; }
}
```

9.3. Function try blocks

So instead the function try block can be wrapped around:

```cpp
#include <iostream>  // std::cout,std::runtime_error,std::exception,std::cerr
class Superclass {
 public:
  Superclass(int x) : x(x) { if (x < 0) throw 1; }  // an exception can be thrown here
  int x;
};

class Subclass : public Superclass {
 public:
  // exceptions from A's constructor are now caught here -- but note the throw --- ...
  Subclass(int x) try : Superclass(x) {}
              catch (int) { std::cerr << "Oops, my bad."; throw; }
};

int main() {    // ... instead of here
  try { Subclass sub(-5); } catch (int) { std::cout << "Oops, my bad.\n"; }
}
```

example02.cpp

9.3. Function try blocks

Function try blocks _for constructors_ are limited: They *cannot* resolve the thrown exception:

```cpp
class Subclass : public Superclass {
 public:
  // exceptions from A's constructor are now caught here -- but note the throw --- ...
  Subclass(int x) try : Superclass(x) {}
                   catch (int) { std::cerr << "Oops, my bad."; throw; }
};
```

- Once the end of the catch block is reached, exceptions will be implicitly re-thrown
- *Other* methods and destructors can throw, rethrow, or resolve the current exception via a return statement
- Reaching the end of the catch block will:
  - implicitly resolve the exception for void-returning functions, and
  - produces undefined behavior for value-returning functions (hence: avoid).

9.4. **noexcept**

Exception handling comes at a (mostly small) cost.
Since C++11, **noexcept** can be added for a class method or function declaration,
to specify that the function could throw exceptions or not:

```
int funct() noexcept;   // funct() does not throw (same as noexcept(true))
void (*fp)() noexcept(false);   // fp points to a function that may throw
```

This allows the compiler to optimize the performance, by skipping the processes that
do exception handling, thus resulting in faster execution of the program.

The **noexcept** *operator* can be used since C++11 to check, at compile time, whether an
expression is declared to not throw any exceptions:

```
noexcept( funct() );   // returns true, see the function declaration above
```

9.4. **noexcept** -- Example

example03.cpp

```cpp
#include <iostream>  // use of std::cout, std::cerr

int divBy(int a, int b) {  // divBy could throw exceptions (noexcept omitted)
  if (b == 0)
    throw std::runtime_error("Error: Division by zero");
  return a / b;
}

int safeDivBy(int a, int b) noexcept {  // safeDivBy won't throw exceptions (noexcept)
  if (b == 0) {
    std::cerr << "Division by zero in safeDivBy\n";
    std::terminate();
  }
  return a / b;
}

int main() {
  std::cout << "divBy: " << noexcept(divBy(7, 0)) << '\n';        // → "divBy: 0"
  std::cout << "safeDivBy: " << noexcept(safeDivBy(7, 0)) << '\n';  // → "safeDivBy: 1"
}
```

9.5. `std::nested_exception` and `std::throw_with_nested`

In C++11 and beyond: Nesting exceptions allow to recursively stack exceptions, generated at the point of the error, without runtime overhead.

```cpp
#include <iostream>  // std::cout
#include <fstream>   // std::ifstream

void run();  // catch exception + wrap it in nested exception
void open_file(const std::string & s);  // catch exception + wrap it

// Nested exception adds 'level' spaces + prints messages via recursion & polymorphism
void print_exception(const std::exception & e, int level = 0) {
  std::cerr << std::string(level, ' ') << "exception: " << e.what() << '\n';
  try { std::rethrow_if_nested(e); }
  catch (const std::exception& nestedException) {
    print_exception(nestedException, level+1);
  }
}
```

example04.cpp

9.5. `std::nested_exception` and `std::throw_with_nested`

```cpp
int main() {  // runs run() and prints the caught exception
  try { run(); } catch (const std::exception & e) { print_exception(e); }
}


void run() {  // catch exception + wrap it in nested exception
  try { open_file("nonExistentFile.txt"); }
  catch (...) { std::throw_with_nested(std::runtime_error("run() fail")); }
}

void open_file(const std::string & s) {  // catch exception + wrap it
  try {
    std::ifstream file(s);  // open file and create an IO fail on next line:
    file.exceptions(std::ios_base::failbit);  // raise exception here
  }
  catch (...) {
    std::throw_with_nested(std::runtime_error("file error: " + s));
  }
}
```

10.1. Streams

10.2. Container Classes

10.1. Streams

A stream class is a class which provides input and output functionality, as a standard abstraction across devices where input and output operations are performed. A stream can be represented as a source or destination, of characters of indefinite length.

For examples sending characters to and receiving characters from …
        disk files, the keyboard and the console, a network connection

10.1. Streams

C++ provides the standard (std::) **iostream** library:
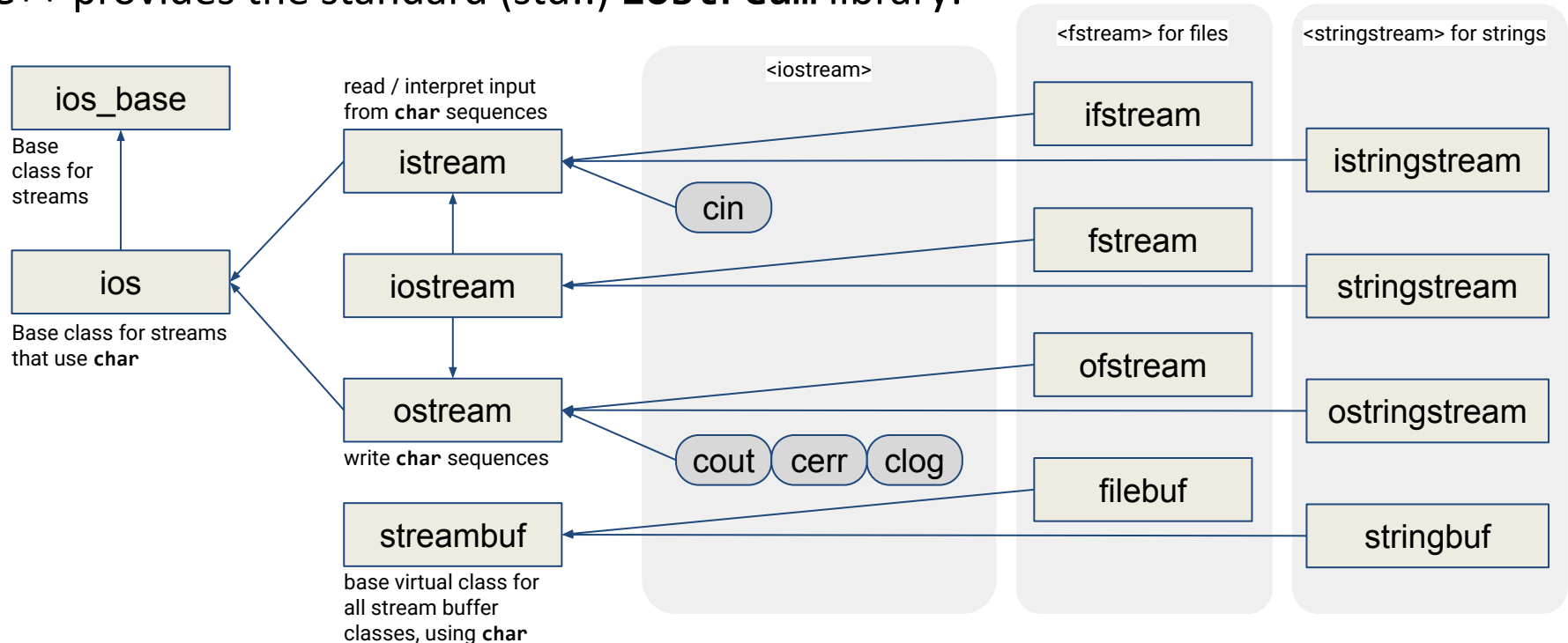
10.1. Streams

`std::streambuf` buffers manage the actual buffer on a lower level, with methods such as **`overflow()`** to handle output (writing), **`underflow()`** to handle input (reading), or **`sync()`** to flush data.

`std::iostream` Streams wrap a **`std::streambuf`** buffer, providing several common methods and operators, including:

- **`get`** / **`put`** a single character from/to a stream before returning
- **`read`** / **`write`** a certain amount of data from/to a stream before returning
- **`getline`** reads characters from an input stream until a delimiter character (usually **`'\n'`** is found) and places them into a string
- stream insertion operator **`<<`** for output to the stream
- stream extraction operator **`>>`** for input to the stream

10.1. Streams

Example 00 (difficulty level: 🌶️🌶️🌶️): Install [boost](#) and compile:

```cpp
/* change the following code to output the server reply straight to a local html file
   and add exception handling in case of file or connection problems */
#include <fstream>
#include <boost/asio.hpp>

int main() {
  char reply[4096];
  boost::asio::ip::tcp::iostream socket("www.example.com", "http");  // socket stream
  std::ofstream outputFile("myTest.txt");  // stream to output file
  socket << "GET / HTTP/1.1\r\nHost: www.example.com\r\nConnection: Close\r\n\r\n";
  socket << std::flush;
  socket.read(reply, 4096);
  outputFile << "Reply of server:\n" << reply;  // output reply of server to text file
}
```

10.1. Streams

Example 01 (difficulty level: 🌶️🌶️🌶️):

```cpp
/* An example of dynamically transforming output (similar to filters in Boost) */
#include <iostream>
class Uppercase : public std::streambuf {  // Class for a custom stream buffer
  std::streambuf* target;
 public:
  Uppercase(std::streambuf* buf) : target(buf) {}
 protected:
  // overflow is virtual std::streambuf function, called by stream writing a char:
  int overflow(int ch) override {
    if (ch != EOF) ch = std::toupper(ch); return target->sputc( ch );
  }
};
int main() {
  Uppercase u( std::cout.rdbuf() );  // buffer links to cout
  std::ostream upperOutput( &u );  // output stream using u
  upperOutput << "this should be uppercase\n";
}
```

10.1. Streams

Example 02 (difficulty level: 🌶️🌶️🌶️):

```cpp
/* Implement the two override methods of RLEstreambuf to implement a basic run
   length encoding compression of a string being sent to an output stream. The
   output of the program below should give "6A5B2CD2A6C3A".
   Note that EOF is a special terminator character that should not be handled.
   Use sputc() to write characters, and '0'+count to write the ascii digits. */
#include <iostream>

class RLEstreambuf : public std::streambuf {  // Class for a custom stream buffer
 public:
  RLEstreambuf(std::streambuf* dest) : dest(dest), last_char(EOF), count(0) {}
 protected:
  int overflow(int ch) override;  // override overflow to handle the compression
  int sync() override;            // flush last data (last character) when done
 private:
  std::streambuf* target;
  int last_char;          // previous character
  int count;              // count of consecutive same characters
};
```

10.1. Streams

Example 02 (difficulty level: 🌶️🌶️🌶️):

```cpp
int main() {
  RLEstreambuf rleBuf(std::cout.rdbuf());
  std::ostream rleOut(&rleBuf);
  std::string input = "AAAAAABBBBBCCDAACCCCCCA";  // Input string to compress
  rleOut << input;  // the output is automatically RLE compressed
  rleOut.flush();    // flush leads to sync being called to output last character
  std::cout << '\n';
}
```

10.2. Container Classes

A container class is a class which implements a data structure containing objects of other classes, with well-defined access patterns (e.g., inserting, finding, removing, or sorting objects), independent of the type of objects stored inside.

Examples: Array, Stack, Queue, List, Tree

10.2. Container Classes

Illustration of a container: A queue of predefined size for integers

```cpp
class Queue {  // Class for a queue of integers
 public:
  Queue(int size = 100) : maxSize(size), tail(0), head(0), filled(0) { items = new int[size]; }
  ~Queue() { delete[] items; items = nullptr; };
  void put(int data);
  int get();
  bool isFull() const { return filled == maxSize; }
  bool isEmpty() const { return filled == 0; }
  void clear() { filled = 0; head = 0; tail = 0; }  // clear whole queue
 private:
  int * items;    // array of integers
  int maxSize;    // size of items
  int tail;       // position in array to put
  int head;       // position in array to get from
  int filled;     // number of elements in queue
};
```
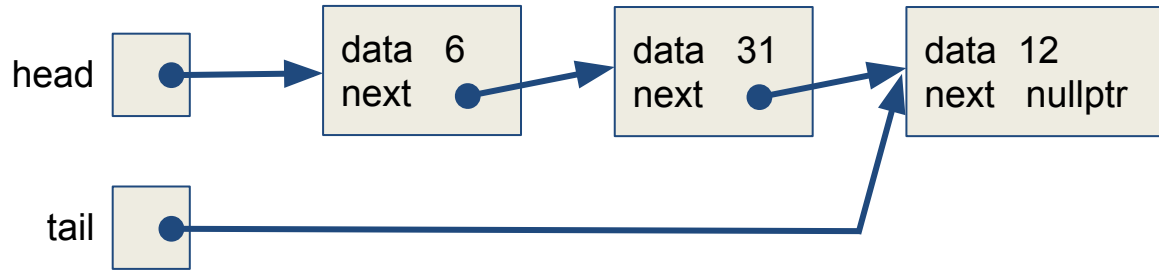
10.2. Container Classes

Illustration of a container: A queue of predefined size for integers

```cpp
// put element at the tail of the queue, for example put(17) updates:
// items:  [  ][  ][  ][  ][  ][17][  ] … [  ][  ]
//                head          tail→
void Queue::put(int data) {   // put element at tail
  if (!isFull()) {
    items[tail] = data;
    tail = (tail+1) % maxSize;
    filled++;
  } else {
    throw std::runtime_error("queue: full on put");
  }
}
```

10.2. Container Classes

Illustration of a container: A queue of predefined size for integers

```cpp
// gets element at the head of the queue, for example get() updates:
// items:   [   ][   ][   ][   ][   ][   ][   ] … [   ][   ]
//                head→                tail
int Queue::get() {  // get and remove element from head
  int retval;
  if (!isEmpty()) {
    retval = items[head];
    head = (head+1) % maxSize;
    filled--;
  } else {
    throw std::runtime_error("queue: empty on get");
  }
  return retval;
}
```

10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```
UQueue q;
q.put(6);
q.put(31);
q.put(12);
```
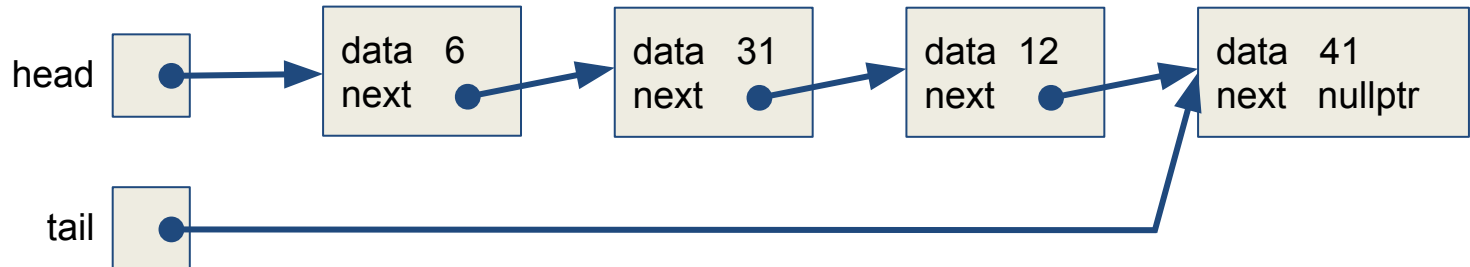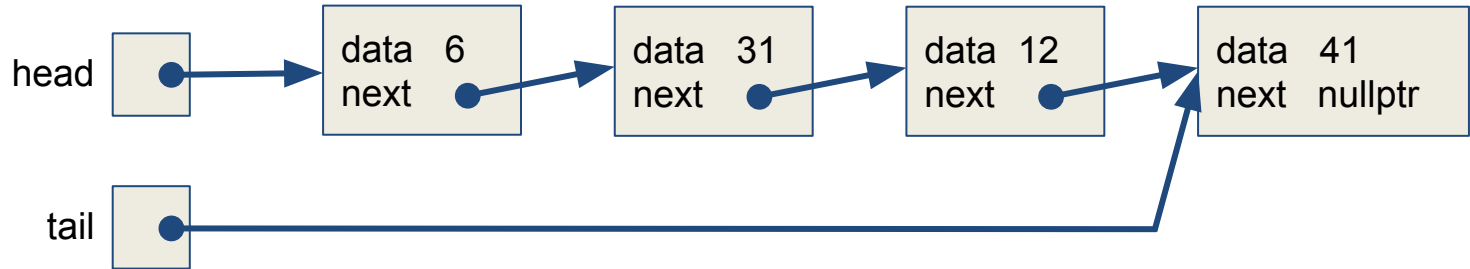
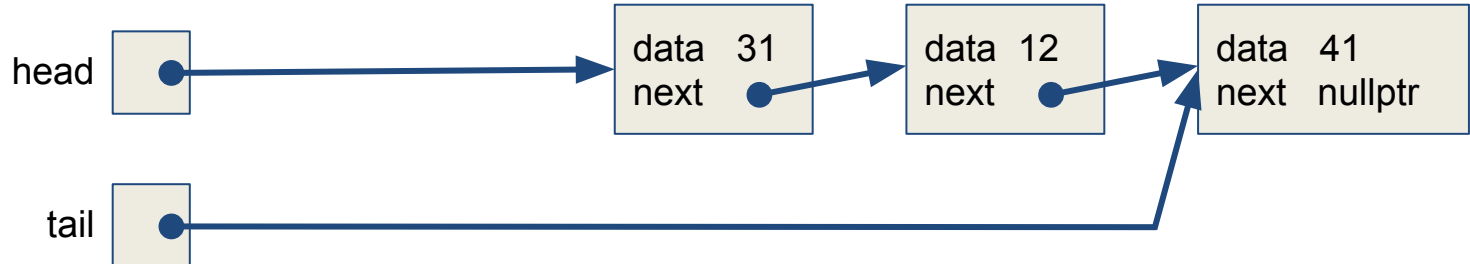head → data 6 / next → data 31 / next → data 12 / next nullptr

tail → data 12

```
q.put(41);
```

head → data 6 / next → data 31 / next → data 12 / next → data 41 / next nullptr

tail → data 41

10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

`q.put(41);`

head → data 6 / next → data 31 / next → data 12 / next → data 41 / next nullptr

tail → data 41 / next nullptr

```
int v;
v = q.get();
```

head → data 31 / next → data 12 / next → data 41 / next nullptr

tail → data 41 / next nullptr

10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```cpp
class QueueElement;  // declaration of the element class

class UQueue {  // Class for an unlimited queue of QueueElements
 public:
  UQueue() { head = tail = nullptr; }
  ~UQueue() { clear(); };
  void put(int data);
  int get();
  bool isEmpty() const;
  void clear();
 private:
  QueueElement * head;  // pointer to element to put
  QueueElement * tail;  // pointer to element to get from
};
```

10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```cpp
class QueueElement {  // element class, hidden from users
 public:
  QueueElement(int data) : data(data) , next(nullptr) {}
  int data;
  QueueElement * next;
};

void UQueue::clear() {  // iteratively clear the queue of all elements
  QueueElement * elem, * elem_next;
  for (elem = head; elem != nullptr; elem = elem_next) {
    elem_next = elem->next;
    delete elem;
  }
  head = tail = nullptr;
}
```

10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```cpp
bool UQueue::isEmpty() const {  // check whether the queue is empty
  return head == nullptr;
}

void UQueue::put(int data) {  // put in new data element at tail
  QueueElement * node = new QueueElement(data);
  if ( isEmpty() ) {
    head = tail = node;
  } else {
    tail = tail->next = node;  // tail is guaranteed to be valid
  }
}
```

10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```cpp
int UQueue::get() {  // get and remove element from head
  int retVal;
  if ( !isEmpty() ) {
    retVal = head->data;
    QueueElement * second = head->next;
    delete head;
    head = second;
  } else {
    throw std::runtime_error("uqueue: empty on get");
  }
  return retVal;
}
```

10.2. Container Classes: Sequence Containers

Sequence Containers maintain the order of elements:

- **std::vector** for dynamic arrays. Use this when you need fast access of any element, and efficient append operations at end of the vector.
- **std::deque** for double-ended queues. Use this when you need fast inserts and removals at both ends of the queue.
- **std::list** for doubly-linked lists. Use this for fast insertions and deletions anywhere in the list.
- **std::forward_list** for singly-linked list. Use this for light-weight lists that only need forward iterations.
- **std::array** Fixed-size array on the stack (no heap). Use this when the size is known at compile time.

10.2. Container Classes: Associative Containers

Associative Containers maintain sorted key-based element collections:

- **std::set**   for sorted, unique elements. Use this when elements are unique and must remain ordered.
- **std::multiset**   for any sorted elements. Use this when elements can be duplicates and need to be ordered.
- **std::map**   for key-value pairs that are sorted by unique keys. Use this when it makes sense to have each element linked to its key.
- **std::multimap**   for key-value pairs that are sorted by keys. Use this when elements can be grouped under the same keys.

10.2. Container Classes: Unordered Associative Containers

Unordered Associative Containers maintain *hash-based* collections:

- **std::unordered_set** for unique elements. Use this when elements are unique and order is not needed.
- **std::unordered_multiset** for any elements. Use this when elements can be duplicates and need to be ordered.
- **std::unordered_map** for key-value pairs, hash-based, with unique keys. Use this when it makes sense to have each element linked to its key and they do not need to be sorted.
- **std::unordered_multimap** for key-value pairs, hash-based. Use this when elements do not need to be ordered and can be grouped under the same keys.

10.2. Container Classes: Container Adapters

Container Adapters are wrappers for other containers to limit their interfaces.
- **std::stack**   for last-in, first-out (LIFO) access of typically **deques**
- **std::queue**   for first-in, first-out (FIFO) access of typically **deques**
- **std::priority_queue**   for elements that are kept in sorted priority order
  (typically using **vector** with heap)

10.2. Container Classes

Example of a container: **`std::multiset`**

```cpp
#include <iostream>
#include <set>
int main() {
  std::multiset<double> grades;  // the multiset to record grades
  // inserting student grades in any order, with duplicates:
  grades.insert(1.7); grades.insert(2.3);  grades.insert(1.0);
  grades.insert(2.3); grades.insert(5.0);  grades.insert(2.3);
  grades.insert(4.0); grades.insert(1.0);  grades.insert(5.0);
  std::cout << "all grades sorted:\n";
  for (auto grade : grades) std::cout << grade << ' ';
  // count how often certain grades occured:
  std::cout << "\ngrade 5.0 was given " << grades.count(5.0) << " times.";
  std::cout << "\ngrade 2.3 was given " << grades.count(2.3) << " times.";
  // erase all 5.0 grades:
  grades.erase(5.0);
  std::cout << "\nall passing grades:\n";
  for (auto grade : grades) std::cout << grade << ' ';
  std::cout << '\n';
}
```

10.2. Container Classes

Example of a container: **`std::priority_queue`**

```cpp
#include <iostream>
int main() {
  // a package has a weight and destination, deliveries are a priority queue
  using Package = std::tuple<double, std::string>;
  // Comparator for max-heap (heavier packages have higher priority):
  auto cmp = [](const Package & a, const Package & b) {
      return std::get<0>(a) < std::get<0>(b);  // index-based access
  };
  std::priority_queue<Package, std::vector<Package>, decltype(cmp)> deliveries(cmp);
  // add deliveries, duplicates possible:
  deliveries.push({2.5, "Berlin"});  deliveries.push({1.2, "Siegen"});
  deliveries.push({3.1, "Berlin"});  deliveries.push({2.5, "Hamburg"});
  deliveries.push({4.1, "Cologne"}); deliveries.push({2.5, "Berlin"});
  // emptying the priority queue:
  std::cout << "handling all deliveries (packages sorted by weight):\n";
  while (!deliveries.empty()) {
    auto [weight, city] = deliveries.top();  // structured binding
    std::cout << weight << " kg - " << city << '\n';
    deliveries.pop();
  }
}
```

10.2. Container Classes

comparison.cpp

```cpp
// create three containers, unordered_set, vector, and forward_list:
std::unordered_set<int> u_set(data.begin(), data.end());
std::vector<int> vec(data.begin(), data.end());
std::forward_list<int> f_list(data.begin(), data.end());

std::cout << "Timing element lookup:\n";
benchmark("unordered_set", u_set, queries, [](const auto & c, int x) {
        return c.find(x) != c.end();
    });
benchmark("vector", vec, queries, [](const auto & c, int x) {
        return std::find(c.begin(), c.end(), x) != c.end();
    });
benchmark("forward_list", f_list, queries, [](const auto & c, int x) {
        return std::find(c.begin(), c.end(), x) != c.end();
    });
```

11.1. Motivation

11.2. Templates

11.3. Using templates, inheritance, friends

11.4. Templates and operator overloading

11.5. The Standard Template Library (STL)

11.1. Motivation: Revisiting the Queue class

The concept of a queue is characterized by its interface and behavior, independent of the type (or class) of elements stored inside.
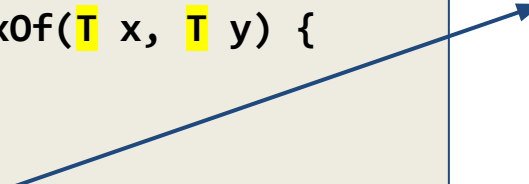
But: The **Queue** and **UQueue** classes in the previous chapter can only store integers; dealing with other types or objects in the queue would require re-writing the **QueueElement** and **Queue** or **UQueue** classes.

Templates allow implementing a container class independent of the type of data elements.

11.1. Motivation: Revisiting the Queue class

Templates are expanded at compile-time, where the compiler does type-checking before the template expansion happens. The source code contains only the function or class, the compiled code can afterwards contain multiple copies of the same function or class. For a function for example:

```cpp
template <class T>  T maxOf(T x, T y) {
  return (x > y)? x : y;
}
int main() {
  std::cout << maxOf<int>(12,24) << ' ';
  std::cout << maxOf<char>('d','e') << '\n';
}
```

```cpp
int maxOf(int x, int y) {
  return (x > y)? x : y;
}
```

```cpp
char maxOf(char x, char y) {
  return (x > y)? x : y;
}
```

11.1. Motivation - Template Queue Example 00

```cpp
                                                              example00.cpp

template <class Data> class Queue {  // Template class for a queue
 public:
  Queue(int size = 100) : maxSize(size), tail(0), head(0), filled(0) {items = new Data[size];}
  ~Queue() { delete[] items; items = nullptr; };
  void put(Data data);
  Data get();
  bool isFull() const { return filled == maxSize; }
  bool isEmpty() const { return filled == 0; }
  void clear() { filled = 0; head = 0; tail = 0; }  // clear whole queue
 private:
  Data *items;      // array of Data
  int maxSize;    // size of items
  int tail;       // position in array to put
  int head;       // position in array to get from
  int filled;     // number of elements in queue
};
```

11.1. Motivation - Template Queue Example 00

`example00.cpp`

```cpp
// put element at the tail of the queue, for example put(d) updates:
// items:  [  ][  ][  ][  ][  ][ d][  ] ... [  ][  ]
//              head           tail→
template <class Data>
void Queue<Data>::put(Data data) {   // put element data at tail
  if (!isFull()) {
    items[tail] = data;
    tail = (tail+1) % maxSize;
    filled++;
  } else {
    throw std::runtime_error("queue: full on put");
  }
}
```

11.1. Motivation - Template Queue Example 00

```cpp
// gets element at the head of the queue, for example get() updates:
// items:  [   ][   ][   ][   ][   ][   ] ... [   ][   ]
//              head→              tail
template <class Data>
Data Queue<Data>::get() {  // get and remove element from head
  Data retval;
  if (!isEmpty()) {
    retval = items[head];
    head = (head+1) % maxSize;
    filled--;
  } else {
    throw std::runtime_error("queue: empty on get");
  }
  return retval;
}
```

11.1. Motivation - Template Queue Example 00

Queue are used while specifying the data type when creating the queue objects:

```cpp
int main() {
  Queue<int> intQueue(127);
  Queue<char> charQueue(21);

  // fill int and char data in both queues:
  for (auto i = 1; i <= 9; i++)  intQueue.put(i);
  for (auto i = '1'; i <= '9'; i++)  charQueue.put(i);

  // get earliest data from both queues nine times:
  for (auto i = 0; i < 9; i++) {
    std::cout << intQueue.get() << ' ' << charQueue.get() << '\n';
  }
}
```

example00.cpp

11.1. Motivation - Template UQueue Example 01

```cpp
template <class Data> class QueueElement;  // element class, hidden from users

template <class Data> class UQueue {    // Template class for an unlimited queue
 public:
  UQueue() { head = tail = nullptr; }
  ~UQueue() { clear(); };
  void put(Data data);
  Data get();
  bool isEmpty() const;
  void clear();
  Data get();
 private:
  QueueElement<Data> * head;  // pointer to element to put
  QueueElement<Data> * tail;  // pointer to element to get from
};
```

example01.cpp

11.1. Motivation - Template UQueue Example 01

example01.cpp

```cpp
template <class Data> class QueueElement {  // element class, hidden from users
 public:
  QueueElement(Data data) : data(data) , next(nullptr) {}
  Data data;
  QueueElement<Data> * next;
};

template <class Data>
void UQueue<Data>::clear() {  // iteratively clear the queue of all elements
  QueueElement<Data> * elem, * elem_next;
  for (elem = head; elem != nullptr; elem = elem_next) {
    elem_next = elem->next;
    delete elem;
  }
  head = tail = nullptr;
}
```

11.1. Motivation - Template UQueue Example 01

example01.cpp

```cpp
template <class Data>
bool UQueue<Data>::isEmpty() const {  // check whether the queue is empty
  return head == nullptr;
}

template <class Data>
void UQueue<Data>::put(Data data) {  // put in new data element at tail
  QueueElement<Data> * node = new QueueElement<Data>(data);
  if (isEmpty()) {
    head = tail = node;
  } else {
    tail = tail->next = node;  // tail is guaranteed to be valid
  }
}
```

11.1. Motivation - Template UQueue Example 01

```cpp
template <class Data>
Data UQueue<Data>::get() {  // get and remove element from head
  Data retVal;
  if (!isEmpty()) {
    retVal = head->data;
    QueueElement<Data> * second = head->next;
    delete head;
    head = second;
  } else {
    throw std::runtime_error("uqueue: empty on get");
  }
  return retVal;
}
```

11.1. Motivation - Template UQueue Example 01

UQueue is used while specifying the data type when creating the queue objects:

```cpp
int main() {
  Queue<int> intQueue(127);
  Queue<char> charQueue(21);

  // fill int and char data in both queues:
  for (auto i = 1; i <= 9; i++)  intQueue.put(i);
  for (auto i = '1'; i <= '9'; i++)  charQueue.put(i);

  // get earliest data from both queues nine times:
  for (auto i = 0; i < 9; i++) {
    std::cout << intQueue.get() << ' ' << charQueue.get() << '\n';
  }
}
```

example01.cpp

11.2. Templates

`template <class Data>` designates Data as the parameter. This is a placeholder for a type to be specified later, when an object of the class that follows this declaration of a template. `class` is <u>almost</u> interchangeable with `typename` .

The later use (e.g., instantiation) of a template is often called specialization:
`UQueue<double> q;` or `Queue<Data> * n = new Queue<Data>(250);`
or `int getSize( Queue<int> & q );`

Templates tell the compiler that the class will use a type that is to be specified when the objects are created, which is why templates are *put in the header file*. The compiler will create the actual class only when it is specialized.

11.3. Using templates, inheritance, friends

Templates can have more than one parameter:

```
template <class XData, class YData>
```

Templates can have, like functions or methods, default parameters:

```
template <class XData, class YData = char>
```

Templates can have non-type parameters, too:

```
template <class XData, int max>
```

Inheritance is possible:

- Templates can inherit from other templates and from non-template classes
  ```
  template <class x> class SortableQueue<x> : public UQueue<x> {
  ```
- Non-template classes can inherit from templates:
  ```
  class IntQueue : public UQueue<int> {
  ```

11.3. Using templates, inheritance, friends

Template parameters for functions can be deduced by the compiler, i.e., without specifying the type. This is possible for template classes since C++17.

```cpp
#include <iostream>                          example06.cpp

template <class T>  T maxOf(T x, T y) {
  return (x > y)? x : y;
}

int main() {
  std::cout << maxOf<int>(12,24) << ' ';
  std::cout << maxOf<char>('d','e') << '\n';
  std::cout << maxOf(12,24) << ' ';       // these two lines have the same
  std::cout << maxOf('d','e') << '\n';  // effect as above, types are deduced
}
```

11.3. Using templates, inheritance, friends

Template classes can declare a non-template friend class or function, a general template friend class or function, or a type-specific template friend class or function:

```cpp
template <class T> class A {  //
 public:
   template<class U> friend class B;
 private:
   T data;
};
template<class U> class B {
 public:
   B() { A<U> a; std::cout << "A's data: " << a.data << '\n'; };
};

int main() {
   B<int> b;
}
```

11.4. Templates and operator overloading

Overloading operators allows to customize the C++ operators (such as **+**, **−**, *****, **/**, **%**, **^**, **&**, **|**, **~**, **!**, **=**, **<**, **>**, **+=**, **−=**, ***=**, **/=**, **%=**, **^=**, **&=**, **|=**, **<<**, **>>**, **>>=**, **<<=**, **==**, **!=**, **<=**, **>=**, **&&**, **||**, **++**, **−−**, **,**, **->***, **->**, **( )**, **[ ]**, **<=>**(since C++20)) for operands of user-defined types. Example:

```cpp
#include <iostream>
class A {
 public:
  char a;
  A(char a): a(a) {}
};
std::string operator + (const A & a1, const A & a2) {
  std::string s;
  s += a1.a; s += a2.a;
  return s;
}
// A a1('a'), a2('b'); std::cout << a1+a2 << '\n';
```

11.4. Templates and operator overloading, reminder: friend methods

Friend methods can also be used for operators:

```cpp
class Complex;
Complex operator + (const Complex & obj1, const Complex & obj2);

class Complex {  // Class representing a complex number, e.g. 2.3 + 4.5 i
 public:
  Complex() : real(0), img(0) {}
  Complex(double real, double imag) : real(real), imag(imag){}
  friend Complex operator + (const Complex & obj1, const Complex & obj2);
 private:
  double real, imag;
};

Complex operator + (const Complex & obj1, const Complex & obj2) {
  Complex temp;
  temp.real = obj1.real + obj2.real;
  temp.imag = obj1.imag + obj2.imag;
  return temp;
}
```

example03.cpp

11.4. Templates and operator overloading, reminder: conversions

Conversion operators can be used to convert one type to another type:

example04.cpp

```cpp
#include <iostream>

// class representing a fraction through numerator and denominator, e.g. 7/9
class Fraction {
 public:
  Fraction(int n, int d) : n(n), d(d) {}
  // note that conversion does not have a return type:
  operator double() const { return double(n)/double(d); }
 private:
  int n, d;
};

int main() {
  Fraction frac(7, 9);
  double val = frac;  // conversion to double, double val = (double) frac;
  std::cout << val << '\n';
}
```

11.4. Templates and operator overloading

An operator with template, now:

```cpp
#include <iostream>                                      example05.cpp

template <class T> class MyClass {  // a very basic template class
 public:
  MyClass(T c) : c(c) {}
  T c;
};

template <class U>
std::ostream & operator<<(std::ostream & out, const MyClass<U> & classObj) {
  out.put(classObj.c);
  return out;
}

int main() {
  MyClass<char> t('?');
  std::cout << t << '\n';
}
```

11.4. Templates and operator overloading

An operator with template, now:

```cpp
#include <iostream>
template <class T> class MyClass;
template <class T> std::ostream & operator<<(std::ostream & out, const MyClass<T> & classObj);

template <class T> class MyClass {  // a very basic template class with a friend operator:
  T c;
 public:
  MyClass(T c) : c(c) {}
  // <> tells C++ this is a friend declaration of a function template instantiated for type T:
  friend std::ostream & operator<< <>(std::ostream & out, const MyClass<T> & classObj);
};

template <class T>
std::ostream & operator<<(std::ostream & out, const MyClass<T> & classObj) {
  out.put(classObj.c);
  return out;
}

int main() {
  MyClass<char> t('?');
  std::cout << t << '\n';
}
```

example06.cpp

11.5. Standard Template Library (STL)

STL is a set of C++ template classes that:

- implement most common data structures for *containers* (queues, vectors, lists, stacks, maps, arrays, etc.), *algorithms* (sorting, search, etc.), *iterators* (to go through containers) , and *function objects or functors*
- in the form of a library of container classes, algorithms, and iterators,
- using components that are parameterized through templates.

11.5. The Standard Template Library (STL) - Vector

Dynamic array that resizes when elements are added or removed

```
std::vector<double> myVector( { 2.0, 4.0, 7.0 } );
std::vector<double> largeVector( 200, 1.0 );  // 200 elements, all ones
std::vector<double> copiedVector( myVector );  // exact copy of myVector
```

```cpp
#include <iostream>
#include <vector>
int main() {
  std::vector<std::string> name( {"John", "George", "Paul", "Smith"} );
  std::cout << "name size=" << name.size() << " of " << name.max_size();
  std::cout << " capacity=" << name.capacity() << '\n';
  for (auto i = name.begin(); i < name.end(); i++)  // use iterators
    std::cout << *i << '\n';
  name.insert(name.begin()+2, "Tom");  // insert another name at third element
  for (auto i : name)  // use range based for loop
    std::cout << i << '\n';
}
```

example07.cpp

11.5. The Standard Template Library (STL) - Map

Container for (key value, mapped value) pairs. Mapped values cannot have the same key values. Initialization can be done (since C++11) with initializer lists:

```cpp
std::map<int, std::string> names = {{1,"Ann"}, {2,"Ames"}, {9,"Asa"}};

std::map<std::string, int> students;
students["Aaron"] = 173923;    // insert two new map elements
students["Zachary"] = 183211;

for (auto const & x : names)  // since C++11
  std::cout << "key:" << x.first << ",val:" << x.second << "\n";

for (auto const & [key, val] : students)  // since C++17
  std::cout << "key:" << key << ",val:" << val << "\n";
```

11.5. The Standard Template Library (STL) - Map

```cpp
#include <iostream>
#include <map>

int main() {
  std::map<std::string, int> students;
  students["Aaron"] = 173923;    // insert new map elements
  students["Zachary"] = 183211;
  students.insert( {"Patrick", 172932} );
  students.insert( std::pair<std::string,int>("Arnold", 161010) );

  for (auto const & x : names)  // since C++11
    std::cout << "key:" << x.first << ",val:" << x.second << '\n';

  for (auto const & [key, val] : students)  // since C++17
    std::cout << "key:" << key << ",val:" << val << '\n';

}
```

example08.cpp

11.5. The Standard Template Library (STL) - transform

```cpp
int increase(int x) { return (x+1); } // operation to execute

int array[] = {1, 2, 3};   // a simple container

 // transform from [ array to array+3 ] to array, with increase:
std::transform(array, array+3, array, increase);

for ( const int & i : array )
  std::cout << i << ", ";  // output: 2, 3, 4,
```

Apart from the unary use above, transform also allows two input containers to be used (e.g., summing up elements of two containers).
A functor can be passed for the given operation to be done on all elements of container, with the additional benefit of keeping its state.

11.5. The Standard Template Library (STL) - transform

```cpp
#include <iostream>                                          example09.cpp
#include <vector>

int main() {
  std::vector a( { 16.0, 17.0, 18.0 } );   // init. lists since C++17
  std::vector b( { 10.0, 30.0, 60.0 } );
  std::vector<double> c;
  double n = 2.7;
  std::transform(a.begin(), a.end(),                // iterators to input and
                 b.begin(), std::back_inserter(c),  // output vector elements
         [=](double x, double y) {return(x - y)/n; });
         // [=] : capture all external variables (n) in lambda by value
         // [&] : capture all external variables (n) in lambda by reference

  for (const double & i : c) {
    std::cout << i << '\n';
  }
}
```

11.5. The Standard Template Library (STL) - functors

Example of a functor:

```cpp
#include <cassert>
class addX {  // this is a functor
 public:
  addX(int val) : x(val) {}  // Constructor sets how much to add
  int operator()(int y) const { return x + y; }  // ()
 private:
  int x;
};

int main() {
  addX add5(5);  // create object of functor class
  int i = add5(3);  // "call" it through the () operation
  assert(i == 8);
}
```

11.5. The Standard Template Library (STL) - functors

More quick examples of functors in standard library:

```cpp
#include <random>
int main() {
  std::random_device r;  // generate uniformly-distributed random integers
  auto n = r();   // calls operator(), returns a random integer
}
```

```cpp
#include <iostream>
int main() {
  std::hash<std::string> h;  // provide hash function for std::string
  std::cout << h("hello") << '\n';  // Output hash value of "hello"
}
```

```cpp
#include <iostream>
int main() {
  std::function<int(int)> square = [](int x) { return x*x; };
  std::cout << square(11) << '\n';  // Outputs 121
}
```

11.5. The Standard Template Library (STL)

```cpp
#include <iostream>
#include <random>  // for generating random values in the data set

int main() {
  // we simulate a sensor data set, rawData, with 1M floats in [0,1023]:
  const size_t dataSize = 1'000'000;
  std::vector<float> rawData(dataSize);
  std::random_device rand;
  std::generate(rawData.begin(), rawData.end(),
        [&]() { return static_cast<float>(rand()) / rand.max() * 1023.0f;} );

  // normalize the data in rawData between 0 and 1, by first finding the maximum
  // value, creating a vector for the normalized data, and then using transform
  // using a lambda function that scales all rawData elements.
  // Bonus: time the transform by using std::chrono::high_resolution_clock::now()

}
```

example10.cpp

11.5. The Standard Template Library (STL)

```cpp
#include <iostream>

// Create a functor called 'Calibrate' for calibration of data, using two
// attributes 'scale' and 'offset', which are used to do the calibration
// through the () operator.

int main() {
  // A time series for temperature values in degrees Celcius:
  std::vector<float> rawTemps = {37.5f, 37.8f, 37.5f, 37.3f, 37.6f};

  // Create a vector 'calibratedTemps' of the same size as rawTemps, and
  // apply Calibrate in std::transform by scaling the temperatur values up
  // by 10% and then shifting the signal down by 2 degrees Celsius

}
```

11.5. The Standard Template Library (STL)

More reference information about the C++ STL is given at these locations:
https://en.cppreference.com/w/cpp/standard_library
https://cplusplus.com/reference/stl/

12.1. Abstract Classes and `virtual`

12.2. The Non-Virtual Interface Idiom

12.3. Multiple Inheritance and the Diamond Problem

12.4. Templated Interfaces

12.5. The Strategy Pattern

12.6. Type Traits and `if constexpr`

12.1. Abstract Classes and `virtual`

Abstract classes are classes that cannot be instantiated and has one or more *pure virtual* (or abstract) methods: `virtual void print() = 0;`

A pure virtual method needs to be overridden by a concrete (i.e., non-abstract) derived class and is indicated in the declaration with the syntax `= 0` behind the method's declaration.

Abstract classes cannot be used as parameter types, as function return types, or as explicit conversion types. Pointers and references to abstract classes can be declared.

12.1. Abstract Classes and `virtual`

```cpp
#include <iostream>                                    Abstract.cpp

class AbstractClass {  // Class has pure virtual method:
 public:
  virtual void printName() const = 0;
 protected:
  std::string name = "myName";   // default initialization since C++11
};

class DerivedClass : public AbstractClass {
 public: // printName is overridden and implemented here:
  virtual void printName() const override { std::cout << name << "\n"; }
};

int main() {
  // This would fail:  AbstractClass myObject;
  DerivedClass myDerivedObject;  // The abstract class forces the
  myDerivedObject.printName();   // implementation of printName
}
```

12.1. Abstract Classes and `virtual`

`virtual` functions or methods can be overridden in derived classes. The overriding is preserved, even if the actual type of the class is not known at compile-time (i.e., when the derived class is handled using a pointer or reference to the base class).

`override` (since C++ 11) can be mentioned after the method declaration, to explicitly show intent to override a method. The compiler can this way stop at programmer's mistakes (for instance, when the method's name was mistyped).

`final` can be mentioned after the method declaration, to explicitly signal that no further subclasses can override this method anymore.

12.1. Abstract Classes and `virtual -- override`

```cpp
#include <iostream>

class BaseClass {
 public:
  virtual void print() const {
   std::cout << "Base Class. \n";
  }
};

class DerivedClass : public BaseClass {
 public:
  virtual void print() const override {
    std::cout << "Derived Class. \n";
  }
};
```

```cpp
int main() {

  BaseClass base; DerivedClass derived;

  BaseClass & bref = base;
  BaseClass & dref = derived;
  bref.print();   // "Base Class."
  dref.print();   // "Derived Class."

  BaseClass * bpnt = &base;
  BaseClass * dpnt = &derived;
  bpnt->print();   // "Base Class."
  dpnt->print();   // "Derived Class."

  bref.BaseClass::print(); // "Base Class."
  dref.BaseClass::print(); // "Base Class."

}
```

12.1. Abstract Classes and `virtual -- final`

```
                                                        Final.cpp
class AbstractClass {  // Class has pure virtual method:
 public:
  virtual void printName() const = 0;
};

class DerivedClass : public AbstractClass {
 public:  // printName is overridden and implemented here:
  virtual void printName() const override { std::cout << "name. \n"; }
};

class FinalClass : public DerivedClass {
 public:  // printName is final-overridden and re-implemented here:
  virtual void printName() const final { std::cout << "Name. \n"; }
};

class AnotherClass : public FinalClass {
 public:  // printName was final in FinalClass, cannot be overridden:
// virtual void printName() const { std::cout << "NAME. \n"; }
};
```

12.1. Abstract Classes and `virtual`

The following code shows that a destructor is not inherited, so objects that are freed in this way do not call the derived class' destructor:

```cpp
#include <iostream>

class BaseClass {  // ~BaseClass illustration:
 public:
  ~BaseClass() { std::cout << " BaseClass resources freed \n"; }
};

class DerivedClass : public BaseClass {  // ~DerivedClass illustration:
 public:
  ~DerivedClass() { std::cout << "DerivedClass resources freed \n"; }
};

int main() {
  BaseClass * base = new DerivedClass;
  delete base;    // " BaseClass resources freed \n"
}
```

12.1. Abstract Classes and `virtual`

Yet, a *virtual* destructor from a base class is always overridden by derived destructors, allowing the following:

```cpp
#include <iostream>

class BaseClass {  // ~BaseClass call is virtual => calls ~DerivedClass
 public:
  virtual ~BaseClass() { std::cout << " BaseClass resources freed \n"; }
};

class DerivedClass : public BaseClass {  // ~DerivedClass afterwards calls ~BaseClass,
 public:                                 // following the typical destructor order
  virtual ~DerivedClass() { std::cout << "DerivedClass resources freed \n"; }
};

int main() {
  BaseClass * base = new DerivedClass;
  delete base;   // "DerivedClass resources freed \n BaseClass resources freed \n"
                 //   ^-- note that now both are called
}
```

12.2. The Non-Virtual Interface Idiom

Remember from Chapter 8: Polymorphism in C++ relies on methods from a base class being declared as **virtual** .

When **Dog** and **Fish** are classes that inherit from **Animal**, objects from these classes have a custom **print()** method, overloading from Animal's **virtual print()** method:

```
Animal * animal;
animal = new Dog("Scooby");
animal->print();  // prints out: I am Scooby. Bark!
animal = new Fish("Salmon");
animal->print();  // prints out: I'm Salmon (fish)
```

12.2. The Non-Virtual Interface Idiom

```cpp
                                                    polyDemo.cpp
#include <iostream>
#include <cstdlib>

class Animal {  // Animal class stores the species and prints this in print()
 protected:
  std::string _species;
 public:
  Animal(std::string species) { _species = species; }
  virtual void print() const { std::cout << "I'm " +  _species << '\n'; }
};

class Dog : public Animal {  // Dogs inherit species from Animal and have a name
 protected:
  std::string _name;
 public:
  Dog(std::string name) : Animal("dog"), _name(name) {}
  void print() const override { std::cout << "I am " << _name << ". Bark!\n"; }
};
```

12.2. The Non-Virtual Interface Idiom

```cpp
                                                    polyDemo.cpp
class Fish : public Animal {  // Fishes have species and subspecies
 protected:
  std::string _subspecies;
 public:
  Fish(std::string subspecies) : Animal("fish"), _subspecies(subspecies) {}
  void print() const override { std::cout << "I'm " << _subspecies << " (fish)\n"; }
};

int main() {
  Animal * animals[4] = { new Dog("Snowy"), new Fish("Salmon"),
                          new Dog("Scooby"), new Animal("some animal")};
  for (auto i=0; i<15; i++) {
    Animal * a = animals[rand() % 4];  // a is a polymorph variable: its
    a->print();                        // print's behavior depends on the
  }                                    // object that a points to
}
```

12.2. The Non-Virtual Interface Idiom

`Animal`'s `virtual print()` method is public. Problems that could occur here are:
- Sub classes do repeat code: The only part that changes is the string to print, but each class needs `std::cout << ... << std::endl;` code
- The base class `Animal` cannot make guarantees about what the `print()` does: Sub-classes may do something completely different as originally intended

This can be fixed by using a **non-virtual interface** that is supplemented by a **private virtual function** that allows polymorphic behaviour. The private virtual methods are called by public non-virtual methods: See next slide

12.2. The Non-Virtual Interface Idiom

```cpp
#include <iostream>                                    polyNVIDemo.cpp
#include <cstdlib>

class Animal {  // Animal class stores the species and prints this in print()
 protected:
  std::string _species;
 public:
  Animal(std::string species) { _species = species; }
  void print() const { std::cout << getSound() << std::endl; }
 private:
  virtual std::string getSound() const { return "I'm " + _species; };
};

class Dog : public Animal {  // Dogs inherit species from Animal and have a name
 protected:
  std::string _name;
 public:
  Dog(std::string name) : Animal("dog"), _name(name) {}
 private:
  std::string getSound() const override { return "I am " + _name + ". Bark!"; }
};
```

12.2. The Non-Virtual Interface Idiom

`polyNVIDemo.cpp`

```cpp
class Fish : public Animal {  // Fishes have species and subspecies
 protected:
  std::string _subspecies;
 public:
  Fish(std::string subspecies) : Animal("fish"), _subspecies(subspecies) {}
 private:
  std::string getSound() const override { return "I'm " + _subspecies + " (fish)"; }
};

int main() {
  Animal * animals[4] = { new Dog("Snowy"), new Fish("Salmon"),
                          new Dog("Scooby"), new Animal("some animal")};
  for (auto i=0; i<15; i++) {
    Animal * a = animals[rand() % 4];  // a is a polymorph variable: its
    a->print();                        // print's behavior depends on the
  }                                    // object that a points to
}
```

12.2. The Non-Virtual Interface Idiom

Thus: Non-Virtual Interfaces decouple a class' public interface (e.g., **Animal**s **print()**) by making it non-virtual, from functions (e.g., **getSound()**) that are providing customization points for sub-classes (e.g., **Dog** or **Fish**).

As an idiom, Non-Virtual Interface is a programming guideline, implementing the Template Method design pattern (not to be confused with C++ templates).

A complete treatise on virtuality can be read in "Virtuality", by Herb Sutter (in C/C++ Users Journal, 19(9), September 2001).

12.3. Multiple Inheritance and the Diamond Problem

Classes often inherit from multiple interfaces (as abstract classes, using multiple pure abstract classes with only pure virtual public methods and static const attributes).

```cpp
class MyInterface {               InterfaceExample.cpp
 public:
  virtual int getFormulaWithX() const = 0;
  virtual ~MyInterface() {};
 public:
  static const int X = 7;
};
```

Classes can in fact also inherit in C++ from multiple base classes in general by separating them with a comma, e.g.:

```cpp
class DerivedClass : public ClassA, public ClassB {
```

In that case, constructors of inherited classes are called in the same order in which they are inherited. The destructors are called in reverse order of the constructors.

12.3. Multiple Inheritance and the Diamond Problem

```cpp
#include <iostream>
class ClassA {
 public:
   ClassA() { std::cout << "Class A constructed.\n"; };
};
class ClassB {
 public:
   ClassB() { std::cout << "Class B constructed.\n"; };
};
class DerivedClass : public ClassA, public ClassB {
 public:
   DerivedClass() { std::cout << "Derived Class constructed.\n"; };
};

int main() {
   DerivedClass myDerivedObject;  // this calls first A's, then B's constructor
}
```
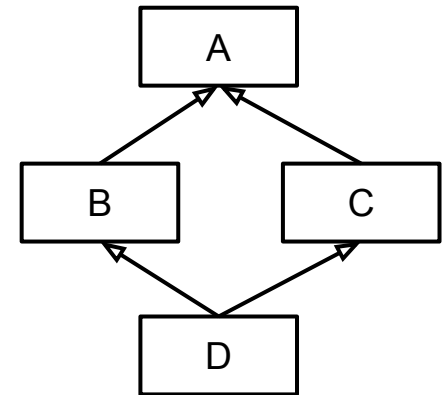
12.3. Multiple Inheritance and the Diamond Problem

The *diamond problem* occurs when two parent classes of a class (class B and class C on the right) have a common child class (D) that inherits from both, and a common parent class (A).

This will lead to the constructor of A being called twice (once via B and once via C). Similarly, the destructor of class A is called twice as well, and objects of class D have two copies of all of A's attributes and methods.

When B and C both inherit a function/method `m()` from A, which is then meant when an object `d` from Class D calls `d.m()` ?

12.3. Multiple Inheritance and the Diamond Problem

```cpp
struct Person {  // Person:
  Person() { std::cout << "Person constructed.\n"; };
  void print() { std::cout << "in print()\n"; }
};
struct Faculty : public Person {  // Faculty is a Person
  Faculty() { std::cout << "Faculty constructed.\n"; };
};
struct Student : public Person {  // Student is a Person
  Student() { std::cout << "Student constructed.\n"; };
};
struct PhDStudent : public Faculty, public Student {  // PhDStudent is both
  PhDStudent() { std::cout << "PhDStudent constructed.\n"; };
};

int main() {
  PhDStudent phd;  // note: this object will contain two copies of a person
  // phd.print();  // error: member found in multiple base-class subobjects
}
```
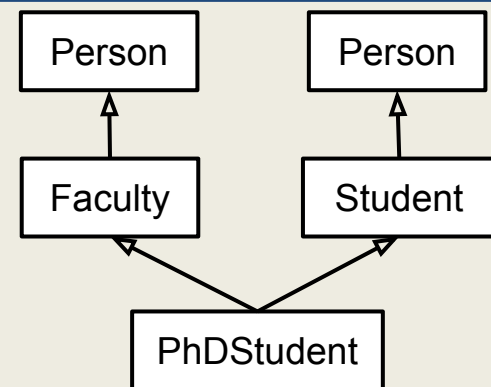
12.3. Multiple Inheritance and the Diamond Problem

```cpp
struct Person {  // Person:
  Person() { std::cout << "Person constructed.\n"; };
};
struct Faculty : virtual public Person {
  Faculty() { std::cout << "Faculty constructed.\n"; };
};
struct Student : virtual public Person {
  Student() { std::cout << "Student constructed.\n"; };
};
struct PhDStudent : public Faculty, public Student {
  PhDStudent() { std::cout << "PhDStudent constructed.\n"; };
};
```
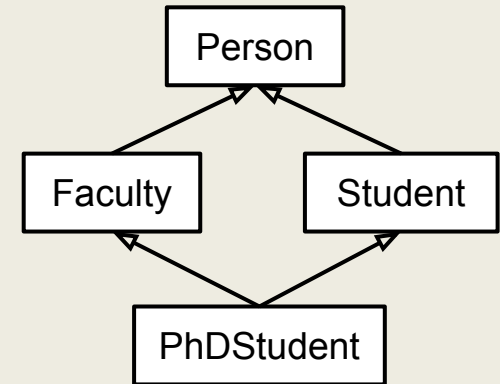
Using *virtual inheritance*, C++ is told that there is one common Person base object for both Faculty and Student and their subclasses (PhDStudent).

12.4. Templated Interfaces

```cpp
#include <iostream>

template <class T>  // force subclasses to
class IMenuItem {   // implement printItem:
 public:
  IMenuItem(T item) : item(item) {}
  virtual void printItem() const = 0;
 protected:
  const T item;     // and hold item here, too
};

template <class T>
class Item : public IMenuItem<T>{
 public:  // implementation of printItem:
  Item(T item) : IMenuItem<T>(item) {}
  void printItem() const override {
    std::cout << "Choice: " << this->item << '\n';
  };
};
```

12.4. Templated Interfaces

```cpp
int main() {  // see next slide for a cleaner version since C++17+
  // menu list where items are given an integer:
  std::array<IMenuItem<int> *, 3> menu
      = { new Item<int>(0), new Item<int>(1), new Item<int>(5)};
  for (auto i = 0; i < menu.size(); i++)
    menu[i]->printItem();
  // menu list where items are given a character:
  std::array<IMenuItem<char> *, 3> menu2
      = { new Item<char>('a'), new Item<char>('b'), new Item<char>('c')};
  for (auto i = 0; i < menu2.size(); i++)
    menu2[i]->printItem();
  // menu list where items are given a string:
  std::array<IMenuItem<std::string> *, 2> menu3
      = { new Item<std::string>(std::string("optionA")),
          new Item<std::string>(std::string("optionB"))};
  for (auto i = 0; i < menu3.size(); i++)
    menu3[i]->printItem();
}
```

12.4. Templated Interfaces

```cpp
int main() {  // with class template argument deduction (C++17+) & range based loops

  // menu list where items are given an integer:
  std::array menu = { new Item(0), new Item(1), new Item(5) };
  for (auto i : menu) i->printItem();

  // menu list where items are given a character:
  std::array menu2 = { new Item('a'), new Item('b'), new Item('c') };
  for (auto i : menu2) i->printItem();

  // menu list where items are given a string:
  std::array menu3 = { new Item(std::string("optionA")),
                       new Item(std::string("optionB")) };
  for (auto i : menu3) i->printItem();

}
```

12.5. Strategy Pattern

The [Strategy Design Pattern](#) defines encapsulated algorithms (strategies) via an interface to make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it through a context.

- The Strategy *interface* declares operations or methods common to all supported versions of some algorithm
- The Strategy *context* uses this interface to call the algorithm defined by concrete Strategies

Example: Imagine having to develop a shopping cart, for which the payment operation can be switched at runtime for each payment (PayPal, Credit Card, etc.).

12.5. Strategy Pattern

```cpp
#include <iostream>                          Strategy.cpp
template <class Currency>
struct PaymentStrategy {  // Strategy Interface
  virtual void pay(Currency amount) = 0;
  virtual ~PaymentStrategy() = default;  // use compiler-generated default
};
// Concrete payment strategies:
template <class Currency>
struct CreditCardPayment : public PaymentStrategy<Currency> {
  void pay(Currency amount) override {
    std::cout << "Paid " << amount << " Euro using Credit Card.\n";
  }
};
template <class Currency>
struct PayPalPayment : public PaymentStrategy<Currency> {
  void pay(Currency amount) override {
    std::cout << "Paid " << amount << " Euro using PayPal.\n";
  }
};
```

12.5. Strategy Pattern

```cpp
// The shopping cart class contains a unique strategy smart pointer, whic
// can be set to any of the above payment strategies. The checkout function
// will then call the strategies' pay method (polymorphism). This is the
// Strategy Context.
template <class Currency>
class ShoppingCart {
  std::unique_ptr<PaymentStrategy<Currency>> strategy;
 public:
  void setPaymentStrategy(std::unique_ptr<PaymentStrategy<Currency>> ps) {
    strategy = std::move(ps);
  }
  void checkout(Currency total) {
    if (strategy) strategy->pay(total);
    else std::cout << "No payment method selected.\n";
  }
};
```

Strategy.cpp

12.5. Strategy Pattern

Strategy.cpp

```cpp
int main() {
  ShoppingCart<double> cart;

  // std::make_unique<CreditCardPayment>() is a function (C++14 and above), which
  // creates a std::unique_ptr smart pointer to a new CreditCardPayment object:
  cart.setPaymentStrategy(std::make_unique<CreditCardPayment<double>>());
  cart.checkout( 123.45 );

  cart.setPaymentStrategy(std::make_unique<PayPalPayment<double>>());
  cart.checkout( 67.89 );
}
```

12.6. Type Traits and `if constexpr`

What if the output of the strategies in the payment example needs to be different (e.g., for `int` vs `double`), or if we want to avoid types (e.g., `char`)?

`if constexpr` (since C++17) enables *compile-time* branching with type traits and can thus avoid generating invalid code for types that don't match the condition, e.g.:

`if constexpr (std::is_integral<Currency>::value)` ensures this branch only compiles for integral types like integers.

`if constexpr (std::is_floating_point<Currency>::value)` ensures this branch only compiles for floating points types such as float or double.
`std::fixed` and `std::setprecision(2)` can then apply monetary formatting.

12.6. Type Traits and `if constexpr`

What if the output of the strategies in the payment example needs to be different (e.g., for **int** vs **double**), or if we want to avoid types (e.g., **char**)?

```cpp
// Concrete Payment Strategies:                    StrategyTypeTraits.cpp
template <class Currency>
struct CreditCardPayment : public PaymentStrategy<Currency> {
  void pay(Currency amount) override {
    std::cout << "Paid ";
    if constexpr (std::is_integral<Currency>::value) {
      std::cout << amount << " (int) Euro";
    } else if constexpr (std::is_floating_point<Currency>::value) {
      std::cout << std::fixed << std::setprecision(2);  // set precision to 2 decimals
      std::cout  << amount << " (float) Euro";
    } else {
      std::cout << " something with an unsupported type";
    }
    std::cout " using Credit Card.\n";
  }
};
```

13.1. Basics of **enum**

13.2. Scoped enumeration **enum class**

13.3. **typedef** and **struct**

13.4. **union** and **std::variant**

13.1. Basics of `enum`

An enumerator (**enum**) creates a data type that can take the value in a set of named integral constants. By default, the first will be **0**, the second **1**, etc.

```
#include <iostream>
int main() {
  enum level_t { LOW, MEDIUM, HIGH };
  level_t danger = HIGH;  // note: HIGH == 2
  std::cout << danger << '\n';
}
```

The integer values that represent the constants can also be set and controlled explicitly. They are numbered in increasing order:

```
enum battery_t { FULL = 100, ADEQUATE = 50, EMPTY = 0 };
enum level_t { LOW = -100, MEDIUM, HIGH };  // MEDIUM == -99, HIGH == -98

enum day_t { MON = 1, TUE, WED, THU, FRI, SAT, SUN};
// MON == 1, TUE == 2, WED == 3, THU == 4, FRI == 5, SAT == 6, SUN == 7
```

13.1. Basics of `enum`

The advantage of **enum** is that the code is readable easier to maintain variables that can take any of a given set of variables :

```cpp
#include <iostream>
int main() {
  enum level_t { LOW, MEDIUM, HIGH };
  level_t humidity = LOW;
  std::string s;
  switch (humidity) {
    case LOW: s = "low"; break;
    case MEDIUM: s = "medium"; break;
    case HIGH: s = "high"; break;
  }
  std::cout << "The humidity is " << s << '\n';
}
```

13.1. Basics of `enum`

Since the values are mapped to integers, this might lead to problems:

```cpp
#include <iostream>
int main() {
  enum level_t { LOW, MEDIUM, HIGH };
  enum battery_t { FULL, ADEQUATE, EMPTY };  // LOW cannot be reused here
  level_t status1 = LOW;       // multiple types' values are basically integers,
  battery_t status2 = EMPTY;   // the following will lead to a warning only:
  std::cout << ( status1 == status2 ) << '\n';
}
```

Typically, **enum**s are used when values are unlikely going to change, such as week days, months, colors, card values.

13.2. Scoped enumeration `enum class`

Since C++11, a *scoped* enumeration (`enum class`) data type is a type-safe enumerator (not a class) that is not implicitly convertible to an integer.

```cpp
enum class Level { LOW, MEDIUM, HIGH };
Level status1 = Level::LOW;  // "status1 = LOW" would lead to error

enum class Battery { FULL, ADEQUATE, EMPTY };
Battery status2 = Battery::EMPTY;  // "status2 = EMPTY" would lead to error

// the following will lead to an "invalid operands" error:
std::cout << ( status1 == status2 ) << '\n';
```

The scoped enumeration's underlying data type can be set explicitly:

```cpp
enum class Choice : int8_t { YES, MAYBE, NO, UNKNOWN };
```

13.2. Scoped enumeration `enum class`

A scoped enumeration cannot be compared to, or use the constants as, integers.

`enum class Level { LOW, MEDIUM, HIGH };` will cause:

`Level l = Level::MEDIUM; std::cout << level;` to result in an error.

The need for scope might also make code less terse and longer:

```cpp
enum class Level { LOW, MED, HIGH };
Level humidity = Level::LOW;
std::string s;
switch (humidity) {
  case Level::LOW: s = "low"; break;
  case Level::MEDIUM: s = "med"; break;
  case Level::HIGH: s = "high"; break;
}
```

Since C++20, `using enum` can import enumerators in the local scope:

```cpp
enum class Level { LOW, MED, HIGH };
Level humidity = Level::LOW;
std::string s;
switch (humidity) {
  using enum Level;
  case LOW: s = "low"; break;
  case MEDIUM: s = "med"; break;
  case HIGH: s = "high"; break;
}
```

13.3. `typedef` and `struct`

Structures (preceded by `struct`) historically combine variables of different types, similar to how arrays combine variables of the same type.

In C++, structures are semantically *very* similar to classes, but by default do not set attributes or methods to private (plus a bit [more](#)).

```cpp
struct anonExamEntry {
 long studentId = 0;  // initialization here
 float grade = 1.0;   // possible since C++11
};

anonExamEntry entry1;
entry1.studentId = 17017491;
entry1.grade = 2.3;
```

13.3. `typedef` and `struct`

`typedef` is a keyword that is used to assign a new name to any existing data-type:
```
typedef int integer; integer i = 9;
```

In C, new variables of a particular structure type would need the keyword **`struct`** to be added each time:
```
struct examEntry {
  long studentId;
  float grade;
};
struct examEntry entry;   // "anonExamEntry entry;" not possible in C
```

Which is why **`typedef`** is used to provide a new name instead:
```
typedef struct examEntry examEntry;   // "struct examentry" = "examEntry"
```

13.4. `union` and `std::variant`

A **union** is a special class type that can hold only one of its non-static attributes.
It is at least as big as needed to store the largest attribute, but is usually not larger.
Unions can have non-virtual methods, but cannot be involved in inheritance.

```cpp
#include <iostream>

union Number {    // a Number has:
  long l;         // either a long,
  unsigned u;     // or an unsigned,
  double f;       // or a double
};
```

union.cpp

```cpp
int main() {
  Number n;
  n.l = 7l;
  std::cout << n.l << '\n';
  n.f = 8.9;
  std::cout << n.f << '\n';
  std::cout << sizeof(n) << '\n';
}
```

13.4. **union** and **std::variant**

Since C++17, STL includes **std::variant**, which replaces many uses of unions and union-like classes:

```cpp
#include <iostream>
#include <variant>

int main() {
  std::variant<char, std::string> s;  // s stores a char or a string
  s = 'a';
  std::visit([](auto x){ std::cout << x << '\n';}, s);  // visit since C++17
  s = "string!";
  std::visit([](auto x){ std::cout << x << '\n';}, s);  // (more info here)
}
```

variant.cpp

13.5. The Visitor Pattern

Visitor allows adding new virtual functions anytime to a family of classes, without modifying these.

Instead, a special visitor class implements all of the appropriate specializations of the virtual function.

This visitor is given the instance reference, and implements the goal through double dispatch (which we can use `std::visit` for).

13.5. The Visitor Pattern

Example: Different user interface events have different properties (position, key, size, etc.), so are represented by different classes.

A Visitor handles all these event types, instead of needing to declare / implement the handling in the Events. Other Visitors for Events can be added later.

```cpp
// Define different event types
struct ClickEvent { int x, y; };
struct KeyEvent { char key; };
struct ResizeEvent { int width, height; };

// An Event is a variant of all possible events
using Event = std::variant<ClickEvent, KeyEvent, ResizeEvent>;
```

Visitor.cpp

13.5. The Visitor Pattern

```cpp
void handleEvent( const Event & event ) {   // Handle Events with a Visitor    Visitor.cpp
  std::visit( [](auto && e) {   // handle in lambda function to std::visit
      using T = std::decay_t<decltype(e)>;   // see explanations in source file
      if constexpr (std::is_same_v<T, ClickEvent>) {
        std::cout << "Click at: " << e.x << ',' << e.y;
      } else if constexpr (std::is_same_v<T, KeyEvent>) {
        std::cout << "Key pressed: " << e.key;
      } else if constexpr (std::is_same_v<T, ResizeEvent>) {
        std::cout << "Window now: " << e.width << 'x' << e.height;
      } else std::cout << "Unknown event.";
      std::cout << '\n';
    }, event);
}
int main() {
  // Create different events, and afterwards handle them through the Visitor:
  Event e[] = { ClickEvent{10, 20}, KeyEvent{'A'}, ResizeEvent{800, 600} };
  for (auto event : e) handleEvent(event);
}
```

14.1. `const` correctness

14.2. `constexpr` – Generalized constant expressions

14.3. Move semantics

14.4. Measuring Time

14.1. `const` correctness

Using **const** tells the compiler that objects/variables should not change:

```cpp
void function1(const std::string & str);    // Pass by reference-to-const

void function2(const std::string * sptr);   // Pass by pointer-to-const

void function3(std::string str);            // Pass by value, str unchanged
```

For the above to-const parameter functions, the C++ compiler checks whether the passed can be changed, or is passed further as a const. Example:

```cpp
void mutate(std::string & s) {};

void function1(const std::string & str) {
  mutate(str);         // compiler error: str is const
  std::string localCopy = str;
  mutate(localCopy);   // fine, localCopy is not const
}
```

14.1. `const` correctness

Declaring the const-ness of a parameter is just another form of type safety and should be done as soon as they are declared.

When you declare something const, *the compiler will treat it as a **different type*** than the non-const version.

const overloading of methods or operators allows const correctness:

```cpp
class Item { /*...*/ };                              constExample01.cpp

class MyItemList {
 public:
   const Item & operator[] (int index) const;  // [] operators often have a
   Item & operator[] (int index);              // const and non-const version
   // ...
};
```

14.1. `const` correctness

`const` correctness allows:

1. protection from accidentally changing variables / objects

2. protection from making accidental variable assignments, e.g.:

```
void myMethod(const int x) {

  if ( x = y )   // typo: really meant if (x == y) -> error
  //  …
}
```

3. the compiler to optimize for it

More examples can be found [here](#).

14.1. `const` correctness

Reminder: `const` pointers can come in various forms, what matters is that everything on the left of the `const` keyword is constant. If `const` is on the full left, what is on its right is constant. `const` pointers need to be directly initialized:

```
int myInteger = 71;
const int constInt = 17;
int const * pointToConstInt = &constInt;    // pointer to const int
int * const constPointToInt = &myInteger;   // const pointer to int
int const * const cPointToCInt = &constInt;  // const pointer to const int
const int * pointToConstInt2 = &constInt;  // pointer to const int
```

14.1. `const` correctness

Reminder: Example 03 from 7. Pointers (difficulty level: 🌶️🌶️🌶️):

```cpp
/** Print a mouse in the console, using a const pointer to avoid changes */
#include <iostream>     // terminal output
[[nodiscard]] auto * getBitmapAddress() {
    static char bitmap[] = "(^._.^)~";  // "bitmap" created in static memory
    return bitmap;  // return pointer to first element
}
int main() {
  // using a pointer to bitmap, and incrementing it, is possible:
  auto * mousePointer = getBitmapAddress();
  while ( *mousePointer != 0 )  std::cout << *(mousePointer++);
  std::cout << "\n";
  // Here mousePointer has changed, it's hard to get the original pointer.
  // Modify the above by protecting the pointer with const and redo the loop.
}
```

14.2. `constexpr` – Generalized constant expressions

Since C++11, the `constexpr` specifier declares that the expression that follows is always evaluated at compile-time, and thus:

- can save potentially significant processing and memory usage during run-time
- but at the cost of more work to be done during compilation

When used to declare variables, these are implicitly `const`s. Example:

```cpp
const int val1 = 3;          // evaluated at compile-time
const int val2 = val1 * 2;   // evaluated at compile-time
int a = 3;                   // variable a is not constant
const int val3 = a;          // evaluated at run-time
constexpr int c1 = val1;     // evaluated at compile-time
// constexpr int c2 = val3;  // compile error, val3 is not constant
```

14.2. `constexpr` – Generalized constant expressions

`constexpr` can precede a function or method. In that case it will be evaluated at compile-time only when all the arguments are evaluated at compile-time:

```cpp
constexpr int square(int value) {
  return value * value;
}
square(4);  // evaluated at compile-time (4 is const)
int val = 4;
square(val);  // evaluated at run-time (val is non-const)
```

If a function has run-time features (e.g., try-catch, assertions*, virtual**, static***, non- constexpr functions, et .), it will be evaluated at run-time.
[*: allowed since C++14 (*), C++20 (**), C++23 (***)]

14.2. `constexpr` – Generalized constant expressions

`constexpr` non-static class methods of run-time objects cannot be used at compile-time if they contain data members or non-compile-time functions:

```cpp
struct Value {
   int val = 3;
   constexpr int getVal() const { return val; }
   static constexpr int get3() { return 3; }
};
```

```cpp
Value v1;
constexpr Value v2;
// constexpr int x = v1.getVal();  // compile error, method not constexpr
constexpr int y = v1.get3();  // same as 'Value::get3()'
constexpr int z = v2.getVal(); // works
```

14.2. `constexpr` – Generalized constant expressions

Since C++17, **`if constexpr`** can be used to compile code on a condition:

```cpp
auto myVersion() {
  if constexpr (__cplusplus == 202101L)  // __cplusplus macro holds c++ version
    return "C++23";   // const char*
  else
    return 11;  // int, returned when c++ version is not 20
}
```

Since C++20, two more keywords can be used:

- **`consteval`** guarantees compile-time evaluation and will produce an error when run-time arguments are supplied
- **`constinit`** guarantees compile-time initialization of variables and will produce an error when run-time arguments are supplied. This is weaker than **`constexpr`**, since the initialized variable *can* change its value later.

14.3. Move semantics

In C++, **the rule of three** is a guideline, which states that if a class defines any of the following three, then it should explicitly define all three:
 (1) destructor, (2) copy constructor, and (3) copy assignment operator
to avoid their default implementation during compilation (which is usually incorrect).

Since C++11, **the rule of five** expands this for these two additional special *move semantics* methods:
 (4) move constructor, and (5) move assignment operator
for the same reason.

More details: https://en.cppreference.com/w/cpp/language/rule_of_three

14.3. Move semantics

Lvalue (Left Value) is something that has a name and a memory address. It can appear on the left-hand side of an assignment and you can take its address with &.

```cpp
int x = 10;      // x is an lvalue
x = 20;          // valid: lvalue on left side
int * p = &x;    // valid: you can take the address of x
```

Rvalue (Right Value) is a temporary value that doesn't have a name or address. It can appear only on the right-hand side of an assignment and you can't take its address directly.

```cpp
int x = 10;
x = 10 + 20;       // 10 + 20 is an rvalue
int y = x * 2;     // x * 2 is an rvalue
//int * p = &(x + 1);  // Error: can't take address of rvalue
```

14.3. Move semantics

Since C++11:

Rvalue references (`Classname &&`) let you bind to rvalues (see move constructors).
`Classname &` binds to lvalues (named objects), `Classname &&` binds to rvalues
(temporary objects). Rvalue references let you "steal" resources from temporary
objects, rather than copying them. This is the foundation of move semantics.
`std::move()` can be used to convert lvalues into rvalues intentionally

```cpp
std::string && strRef = std::string("Hello");  // rvalue reference
                        // to temporary string, strRef is now "Hello"
strRef += ", world";   // strRef can be modified, is now "Hello, world"
std::string s = "hi";  // s is lvalue
// std::string && badRef = s;  // Error: binding rvalue reference to lvalue
```

14.3. Move semantics -- rule of three

Example: Message class

```cpp
                                                    Message_v1.cpp

class Message {
  char * text;
 public:
  Message(const char * str);              // constructor with C string
  ~Message();                             // 1. Destructor
  Message(const Message & other);         // 2. Copy constructor
  Message & operator=(Message & other);   // 3. Assignment operator

  // friend method that returns a reference to a concatenated string:
  friend Message & operator+(const Message & m1, const Message & m2);

  void show() { std::cout << text << '\n'; }
};
```

14.3. Move semantics -- rule of three

Example: Message class

```cpp
                                            Message_v1.cpp

Message::Message(const char * str) {    // copy from str:
  text = new char[std::strlen(str) + 1];
  std::strcpy(text, str);
}

Message::~Message() { delete[] text; }

Message::Message(const Message & other) {  // perform deep copy:
  text = new char[std::strlen(other.text) + 1];
  std::strcpy(text, other.text);
}

Message & Message::operator=(Message & other) {
  std::swap(*this, other);  // see copy-swap idiom
  return *this;
}
```

14.3. Move semantics -- rule of three

Example: Message class

Message_v1.cpp

```cpp
// friend operator that concatenates two Messages:
Message & operator+(const Message & m1, const Message & m2) {
  char * text = new char[std::strlen(m1.text) + std::strlen(m2.text) + 1];
  std::strcpy(text, m1.text);
  std::strcat(text, m2.text);
  Message result(text);
  delete[] text;
  return result;  // return by value or move
}
```

14.3. Move semantics -- rule of three

Example: Message class

```
int main() {
  Message s1("ping!");  // s1 is an object from C string
  Message s2(s1);       // s2's copy constructor from s1: lvalue
  Message s3(s1+s2);    // s3's copy constructor from s1+s2: rvalue?
  Message s4 = s1;      // s4's assignment operator copies
  s1.show(); s2.show(); s3.show(); s4.show();
}
```

Message_v1.cpp

In the above, **s1** as a parameter to s2's copy constructor is an **lvalue**.

(s1+s2) as a parameter for s3's copy constructor *could* be an **rvalue**, a temporary object that is removed after the statement on that line is finished.

If it were an **rvalue**, the move constructor would be called instead of the copy constructor, allowing for better performance: see next slides.

14.3. Move semantics -- rule of five

Example: Message class, *with* rule of five

```cpp
class OwnString {                                   Message_v2.cpp
  char * text;
 public:
 Message(const char * str);                // single constructor with C string
 ~Message();                               // 1. Destructor
 Message(const Message & other);           // 2. Copy constructor
 Message & operator=(const Message & other);  // 3. Assignment operator
 Message(Message && other);                // 4. Move constructor
 Message & operator=(Message && other); // 5. Move assignment operator
 // friend method that returns a Message having a concatenated string:
 friend Message operator+(const Message & m1, const Message & m2);
 // print out the Message object address and text:
 void show() { std::cout << this << ':' << (text?text:"?") << '\n'; }
};
```

14.3. Move semantics -- rule of five

Example: Message class, *with* rule of five

```cpp
int main() {                                    OwnString_v2.cpp
  Message s1("ping!");   // s1 is an object constructed from C string
  Message s2(s1);        // s2's copy constructor from s1: lvalue
  Message s3(s1+s2);     // s3's constructor from s1+s2: rvalue
  Message s4 = s1;       // s4 is copy-constructed here by the compiler
  Message s5("pong!");   // s5 is constructed here
  s5 = s2;               // s5 is assigned here
  Message s6 = std::move(s2);   // s6 is move constructed here
  s1.show(); s2.show(); s3.show(); s4.show(); s5.show(); s6.show();
}
```

14.4. Measuring Time

For measuring how long a program needed to perform a task, there are three types of time measurement:

- **Wall-Clock/Real time**: Human-perceived passage of time from the start to the completion of a task (includes other processes taking resources, too)
- **User/CPU time**: The time spent by the CPU to process user code
- **System time**: The time spent by the CPU to process system calls (including I/O calls) executed into kernel code

14.4. Measuring Time - Wall-clock time

On Linux / MacOSX (resolution in microseconds):

WallClock.cpp

```cpp
#include <time.h>        //struct timeval
#include <sys/time.h>    //gettimeofday()
#include <iostream>

int main() {
  struct timeval start, end;  // struct timeval {second, microseconds}
  ::gettimeofday(&start, NULL);
  double ret = 0;
  for (int i=0; i<0xFFFFF; i++) { ret += ret*0.3; }  // task to be measured
  ::gettimeofday(&end, NULL);
  long start_time = start.tv_sec * 1000000 + start.tv_usec;
  long end_time = end.tv_sec * 1000000 + end.tv_usec;
  std::cout << "Time: " << end_time - start_time << " microsecs.\n";
}
```

14.4. Measuring Time - User time

Using **std::clock** (resolution in nanoseconds):

UserTime.cpp

```cpp
#include <chrono>  // clock_t, std::clock
#include <iostream>

int main() {
  clock_t start_time = std::clock();
  double ret = 0;
  for (int i=0; i<0xFFFFF; i++) { ret += ret*0.3; }  // task to be measured
  clock_t end_time = std::clock();
  float diff = static_cast<float>(end_time - start_time);  // static_cast
  diff /= CLOCKS_PER_SEC;  // POSIX-defined as 1000000
  std::cout << "Time: " << 1000*diff << " milliseconds \n";
}
```

14.4. Measuring Time - User & System time

Using <sys/times.h> (resolution in milliseconds):

UserSystemTime.cpp

```cpp
#include <unistd.h>  // _SC_CLK_TCLK
#include <sys/times.h>  // struct ::tms
#include <iostream>

int main() {
  double ret = 0;
  struct ::tms start_time, end_time;
  ::times(&start_time);
  for (long i=0; i<0xFFFFFFFF; i++) { ret += ret*0.3; }  // task to measure
  ::times(&end_time);
  auto user_diff = end_time.tms_utime - start_time.tms_utime;
  auto sys_diff = end_time.tms_stime - start_time.tms_stime;
  float user = static_cast<float>(user_diff) / ::sysconf(_SC_CLK_TCK);
  float system = static_cast<float>(sys_diff) / ::sysconf(_SC_CLK_TCK);
  std::cout << "User Time:   " << user << " seconds \n";
  std::cout << "System Time: " << system << " seconds \n";
}
```