

4.1. Functions and their parameters

4.2. Recursive Functions

4.3. Call by Value

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

4.5. Default Parameters and Function Attributes

4.6. Header files and Modules

4.7. Miscellaneous: Variadic arguments and Coroutines

## 4.1. Functions and their parameters

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

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

## 4.1. Functions and their parameters: Declaring Functions

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

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

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

## 4.1. Functions and their parameters: Declaring Functions

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

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

## 4.1. Functions and their parameters: Declaring Functions

- Functions can call other functions, allowing cycles:

function **a()** calls **b()**, **b()** calls **a()**

→ In this case, declarations need to come first, example:

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

## 4.1. Functions and their parameters: Declaring Functions

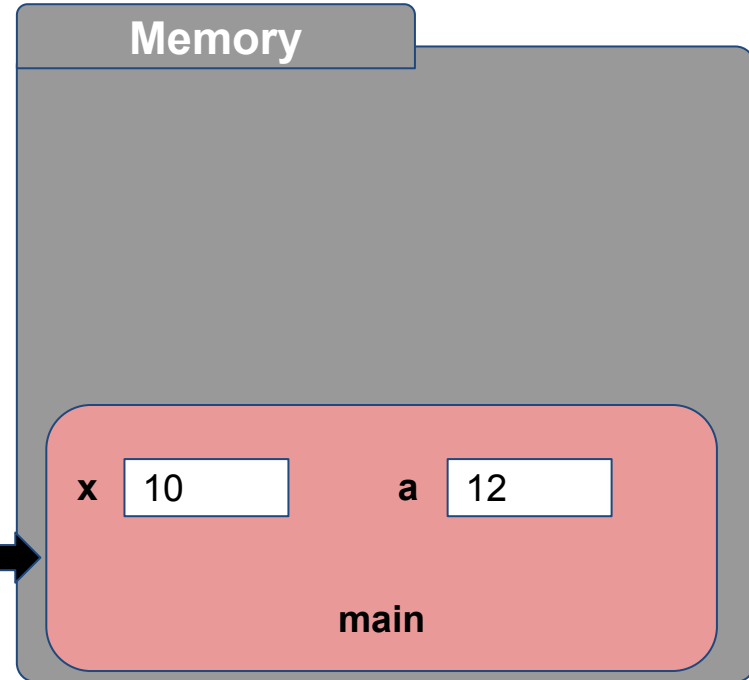
- A function declaration can have *parameters*: variables that obtain a value when the function is called and that are treated as local variables in the implementation of the function
- A function can have a return type. If not, we use `void` → [Is this a type?](#)

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

## 4.1. Functions and their parameters: Using Functions

- A function is *called*:

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

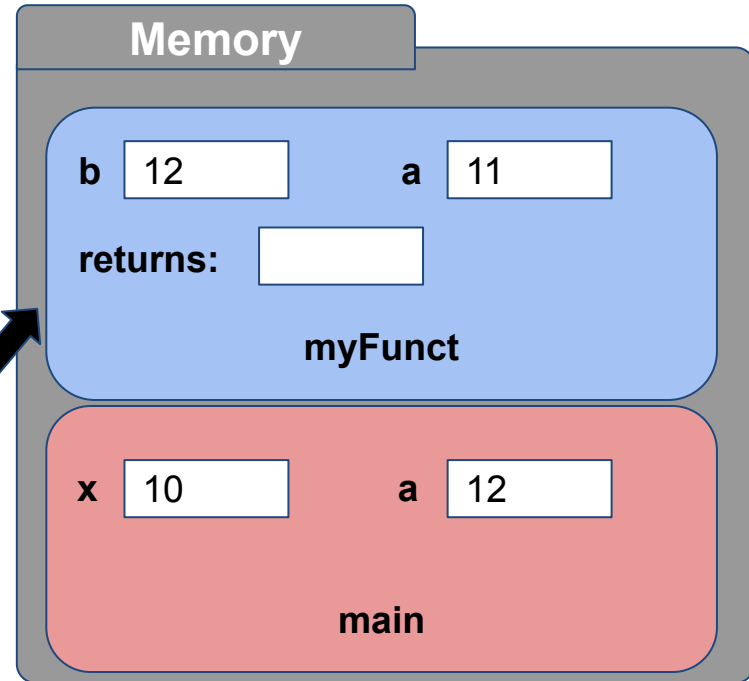


A stack is created in memory, in which the function's local variables are stored

## 4.1. Functions and their parameters: Using Functions

- A function is *called*:

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



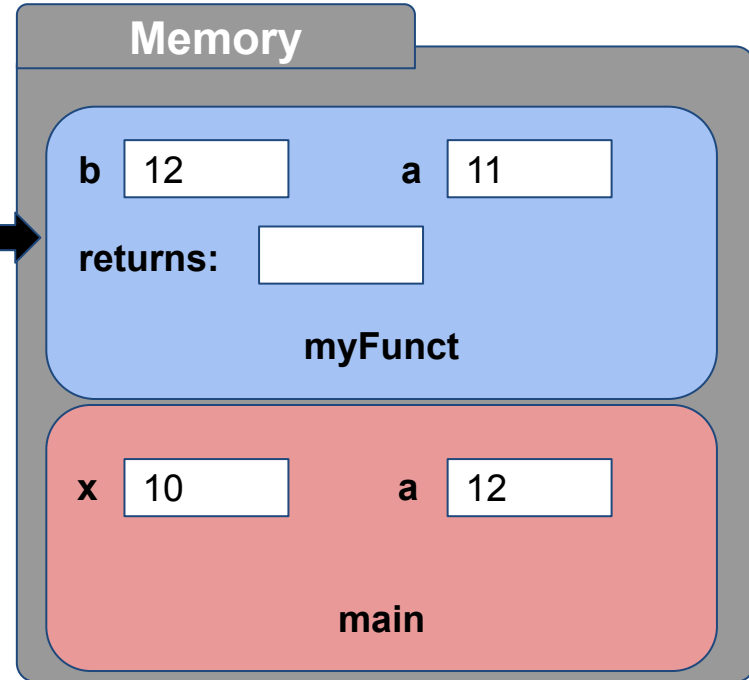
A stack is created in memory, in which the function's local variables are stored



## 4.1. Functions and their parameters: Using Functions

- A function is *called*:

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

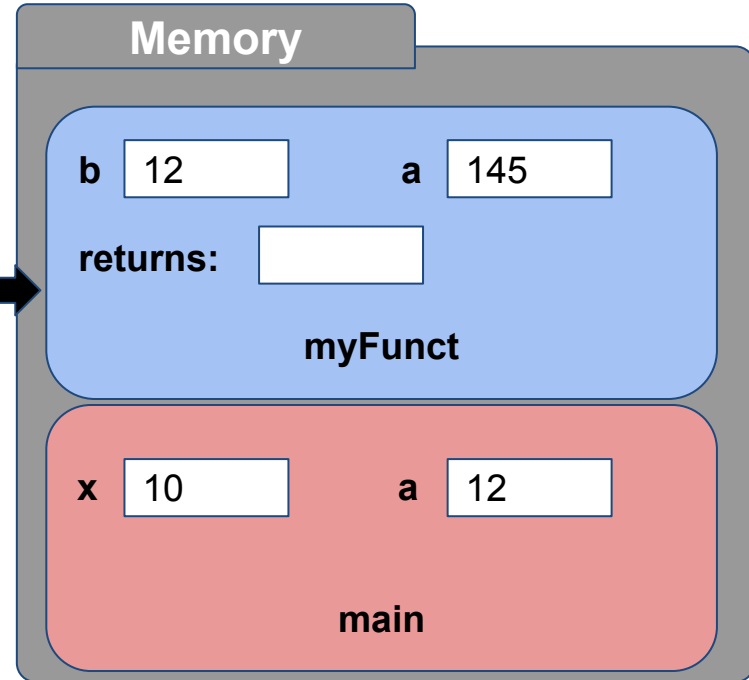


A stack is created in memory, in which the function's local variables are stored

## 4.1. Functions and their parameters: Using Functions

- A function is *called*:

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

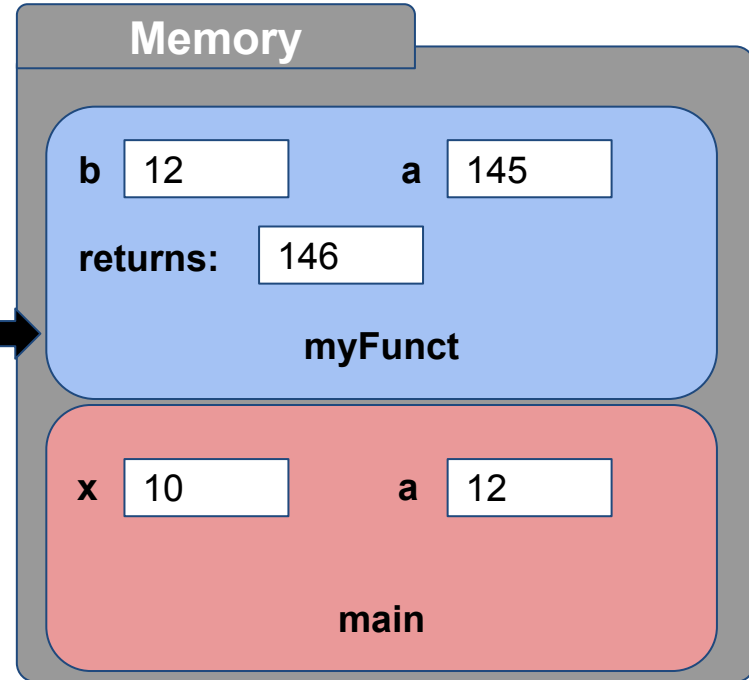


A stack is created in memory, in which the function's local variables are stored

## 4.1. Functions and their parameters: Using Functions

- A function is *called*:

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

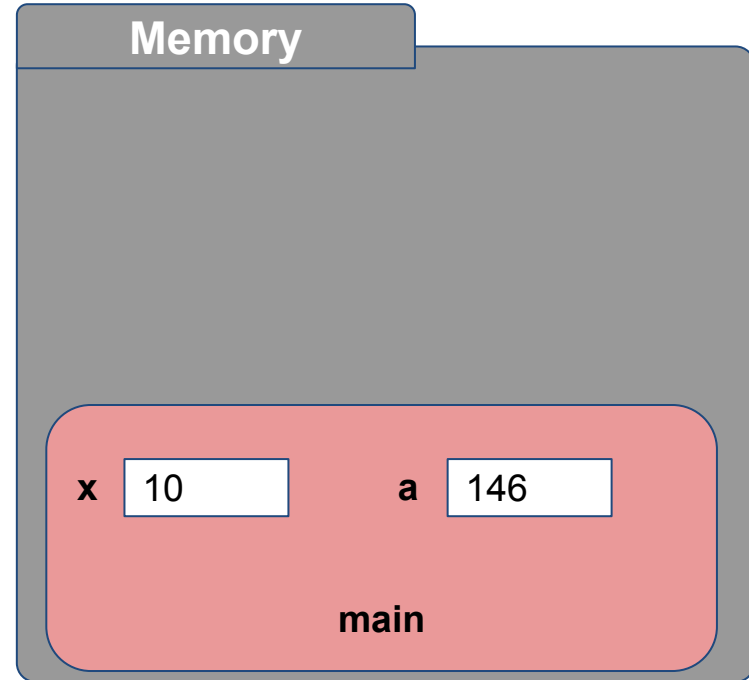


A stack is created in memory, in which the function's local variables are stored

## 4.1. Functions and their parameters: Using Functions

- A function is *called*:

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



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

## 4.1. Functions and their parameters: Using Functions

- Maze Game v.1.0: expand this code to move the player and [add color](#)

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

## 4.1. Functions and their parameters: Using Functions

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

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

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

## 4.1. Functions and their parameters: Using Functions

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

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

## 4.1. Functions and their parameters: Using Functions

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



## 4.2. Recursive Functions

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

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

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

Mathematical definition:

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

so:

$$2! = 2 \cdot 1 = 2$$

$$3! = 3 \cdot 2 \cdot 1 = 6$$

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

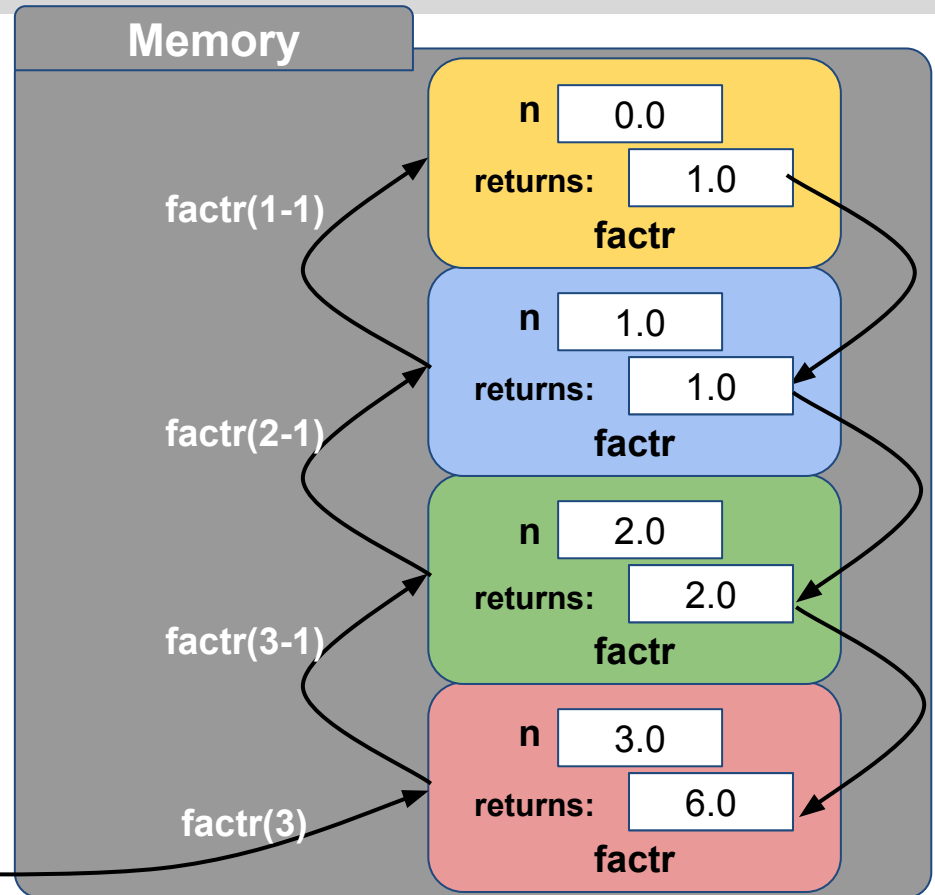
and so on ...

## 4.2. Recursive Functions

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

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

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



### 4.3. Call by Value

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

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

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

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

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

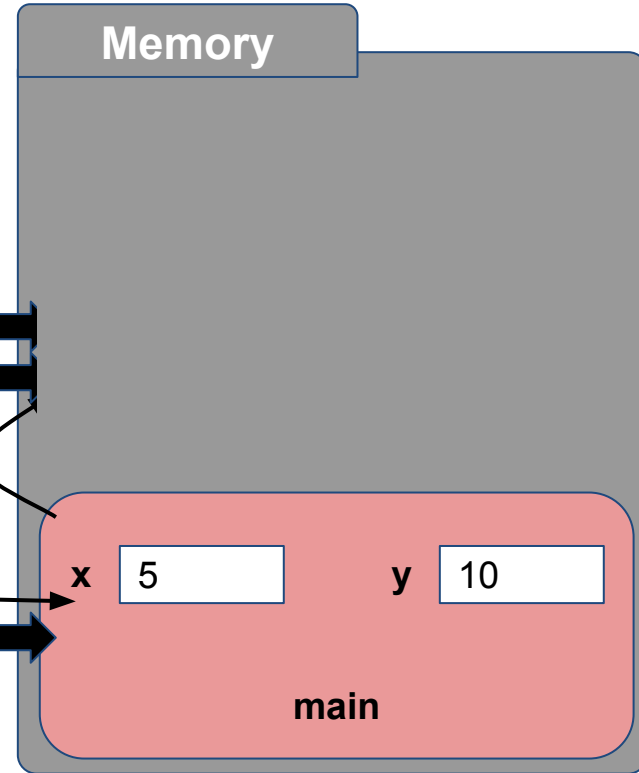
```
int a = 10;  
y = maximum(a, a); // the value 10 is copied to both parameters
```

### 4.3. Call by Value

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

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

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

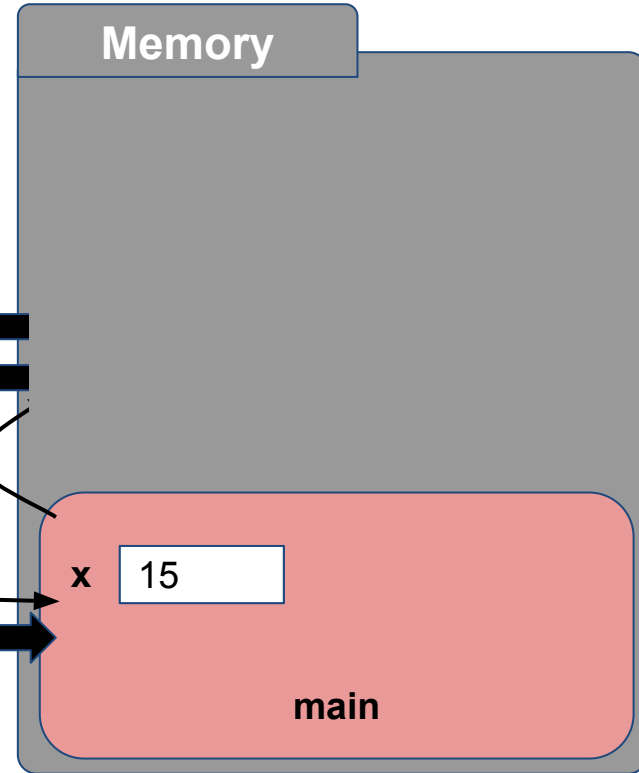


### 4.3. Call by Value

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

You can use the function's return value:

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



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

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

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

## 4.4. inline Functions, Overloading, =delete

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

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

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

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

## 4.4. inline Functions, Overloading, =delete

- There are four Overloading Resolution Rules
  - An exact match between parameter types
  - A promotion (e.g., char to int )
  - A standard type conversion (e.g. float and int )
  - A constructor or user-defined type conversion (see later)
- **= delete** can be used to prevent calling the wrong overload:

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



## 4.5. Default Parameters and Function Attributes

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

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

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

## 4.5. Default Parameters and Function Attributes

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

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

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

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

## 4.6. Header files and Modules

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

## 4.6. Header files and Modules

```
/* 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;
}
```

Maze.cpp

## 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

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

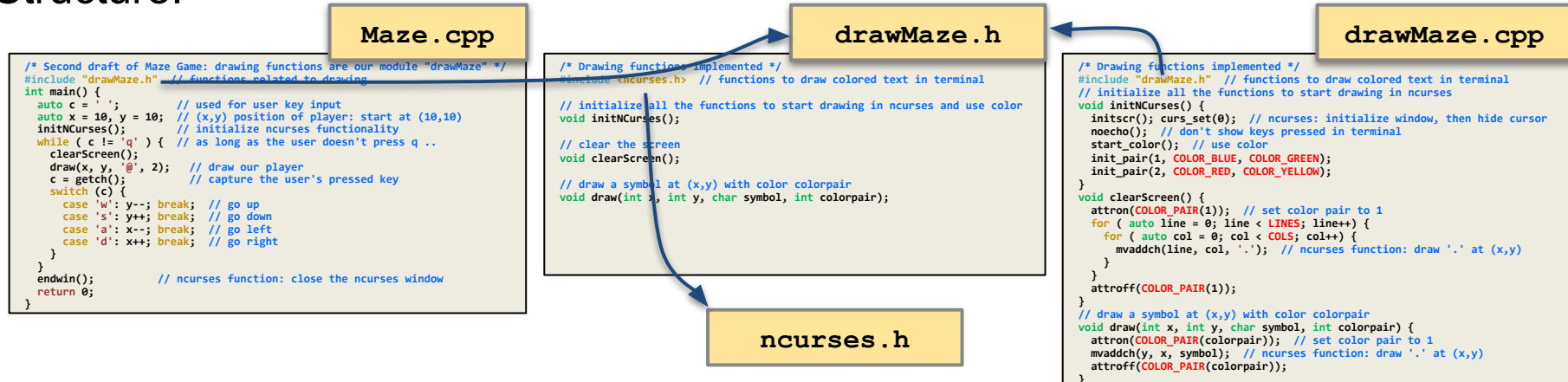
drawMaze.cpp

## 4.6. Header files and Modules

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

drawMaze.cpp

### Structure:



## 4.6. Header files and Modules

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

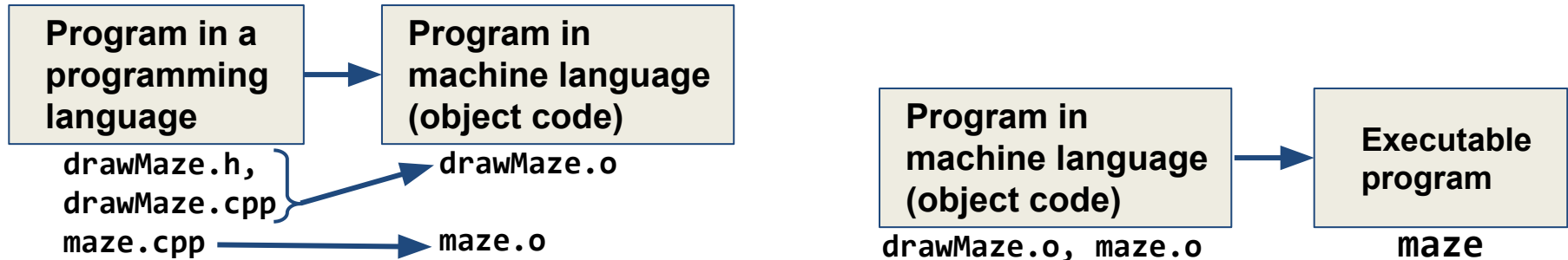
- First compile the module and the program into object files:

```
g++ -c drawMaze.cpp -std=c++11 → object file drawMaze.o
```

```
g++ -c maze.cpp -std=c++11 → object file maze.o is created
```

- Then link the object files:

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



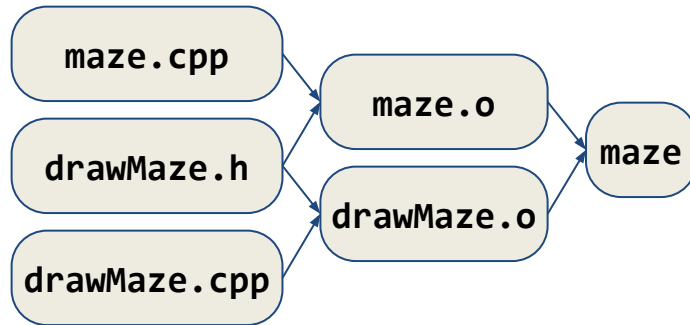


## 4.6. Header files and Modules

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

## 4.6. Header files and Modules: The `make` utility

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



compile `drawMaze.cpp`:

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

compile `maze.cpp`:

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

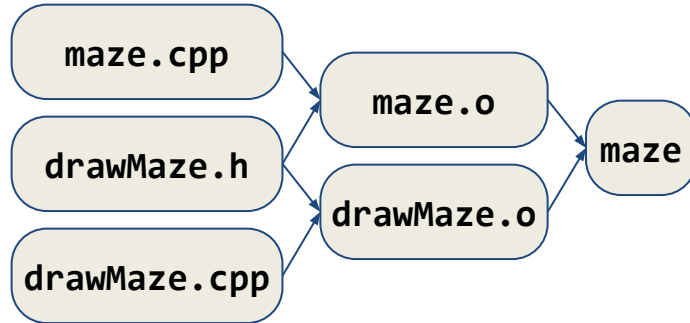
link the objects files into the executable program `maze`:

```
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**



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

**Makefile**

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

## Summary

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

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

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

5.1. Array basics

5.2. Multidimensional Arrays

5.3. Strings (Arrays of char)

5.4. Arrays as function parameters

5.5. Reading char arrays from the terminal

5.6. Lambda Expressions and **foreach** Loops

## 5.1. Arrays: Reminders

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

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

For example:

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

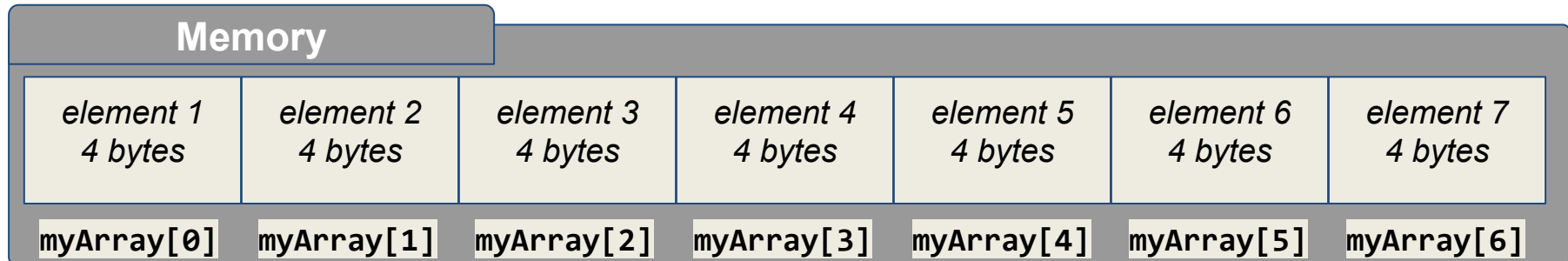
- that 4 bytes need to be reserved for each of them,
- which are organized so they span the whole numbers from -2147483648 to 2147483647
- and that `height * width` is a legal operation

## 5.1. Arrays

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

For example:

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



## 5.1. Arrays: Initialization, sizeof

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

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

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

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

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

- Loops are typically used for larger arrays:

```
bool myArray[400];
```

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

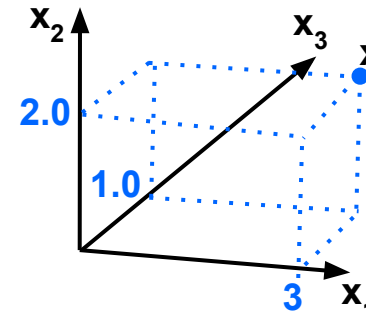
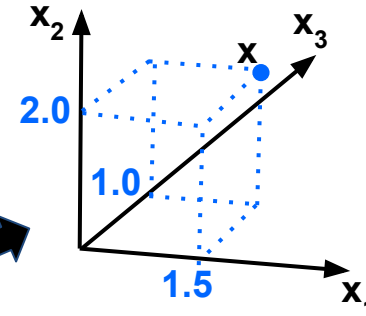


## 5.1. Arrays

- Example: a three-dimensional vector

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

x[0] = 3.0;
```

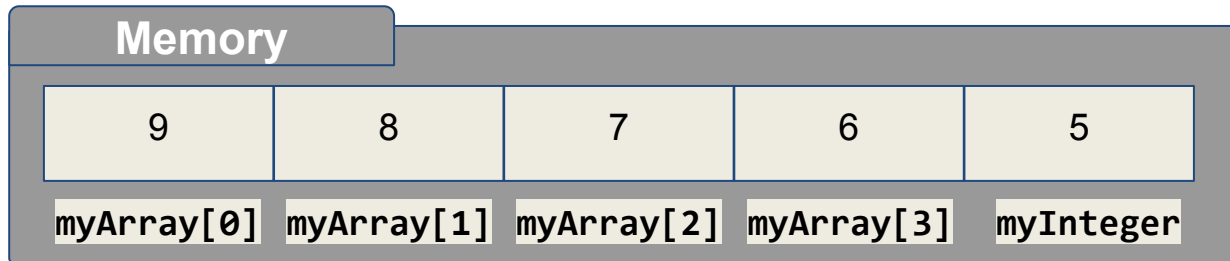


## 5.1. Arrays: Writing beyond the array boundary

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

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

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



## 5.1. Arrays

Example 01 (difficulty level: 🌶️)

```
/**  
    Write a program that initializes an array of 50 booleans, repeatedly having two  
    elements with a true value, followed by one element with false.  
    So the array starts with: true, true, false, true, true, false, true, true, ...  
    Do not use any variables other than myArray.  
*/  
  
int main() {  
    bool myArray[50];  
  
    return 0;  
}
```

## 5.1. Arrays

Example 02 (difficulty level: 🌶️🌶️)

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

    return 0;
}
```



## 5.2. 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  $2 \times 4 \times 4 = 32$  bytes
- `sizeof(myTable[0])` will return  $4 \times 4 = 16$  bytes

## 5.2. Multidimensional Arrays: Maze Game v.3.00

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

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

## 5.2. Multidimensional Arrays: Maze Game v.3.00

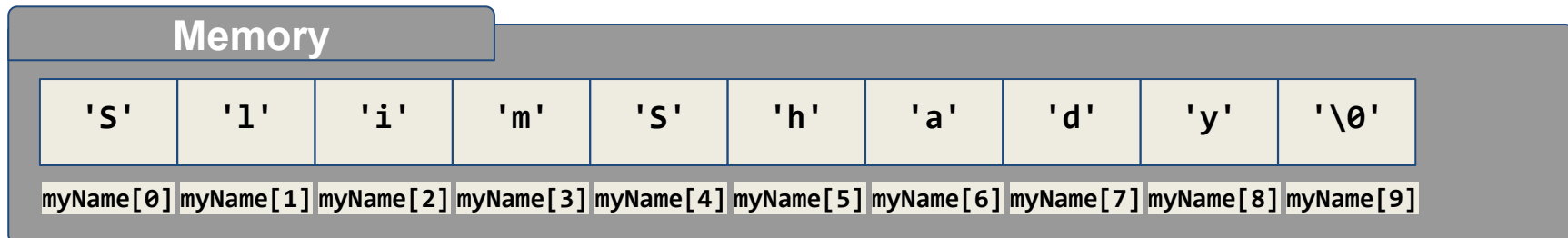
```
/* Third draft of Maze Game: We add an actual maze to our module "drawMaze" */
#include "drawMaze.h" // functions related to drawing the maze and player
int main() {
    auto c = ' '; // used for user key input
    auto x = 10, y = 10; // (x,y) position of player: start at (10,10)
    initNCurses();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



## 5.3. Arrays: Strings (Arrays of char)

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

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



## 5.3. Arrays: Strings (Arrays of char)

- Later, we will see that :

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

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

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

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

## 5.3. Arrays: Strings (Arrays of char)

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

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

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

## 5.4. Arrays as function parameters

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

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

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

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

## 5.5. Reading char arrays from the terminal

- When trying our this approach:

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

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

- The correct approach is to use:

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

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

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

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

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

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

capture clause (see later)

parameters

return type

function body

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

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

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

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

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



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

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

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