4. Functions, Recursion, Call by Value

4.1. Functions and their parameters

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

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

4.1. Functions and their parameters: Declaring Functions

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

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

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

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

4.1. Functions and their parameters: Declaring Functions

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

```
#include <iostream>  // output to the console
#include <cstdint>  // we're using the uint16_t type
void drawLine(char symbol = '-', uint16_t len = 25) {
  for (auto line = 0; line < len; line++) std::cout << symbol;
  std::cout << '\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:

```
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

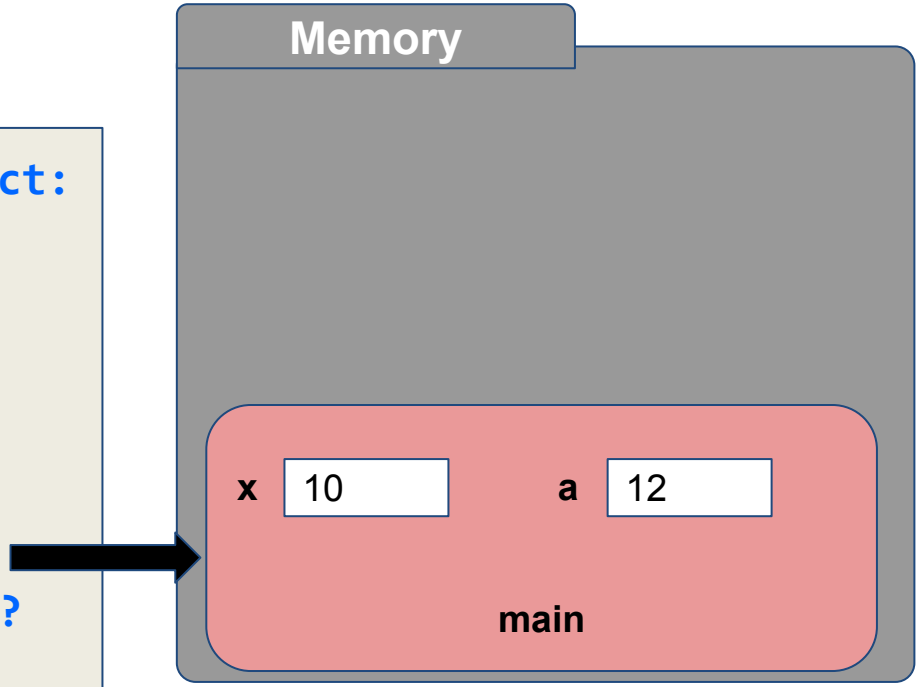
- A function declaration can have *parameters*: variables that obtain a value when the function is called and that are treated as local variables in the implementation of the function
- A function can have a return type. If not, we use `void` → Is this a type?

```
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;
}
```

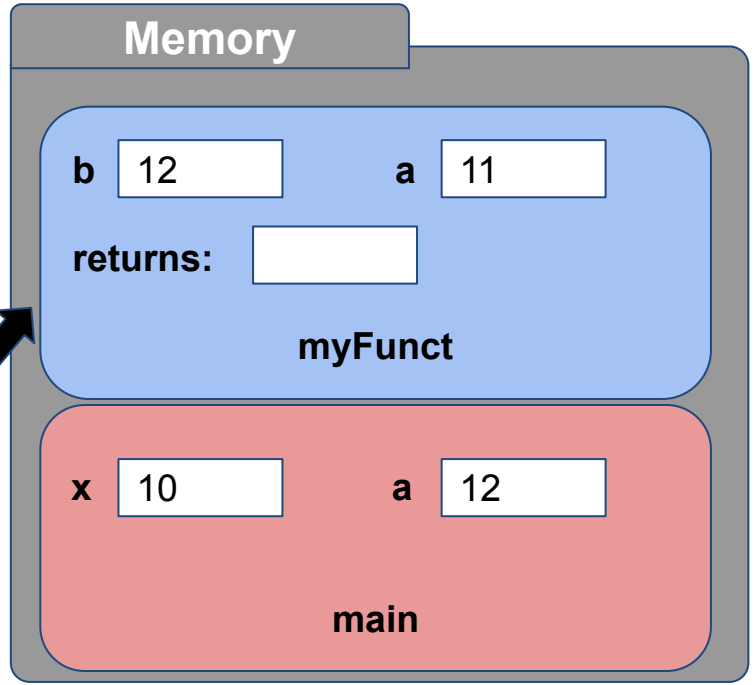


A stack is created in memory, in which the function's local variables are stored

4.1. Functions and their parameters: Using Functions

- A function is *called*:

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

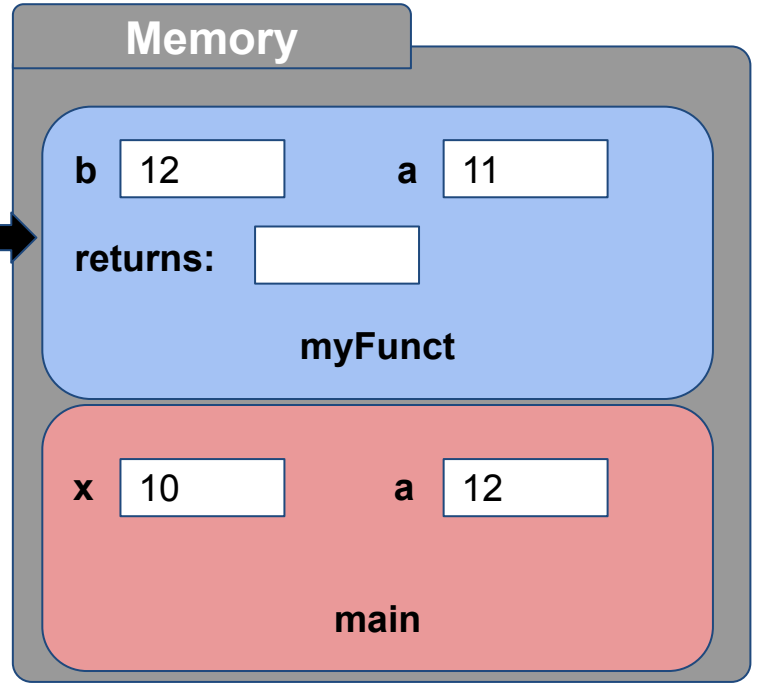


A stack is created in memory, in which the function's local variables are stored

4.1. Functions and their parameters: Using Functions

- A function is *called*:

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

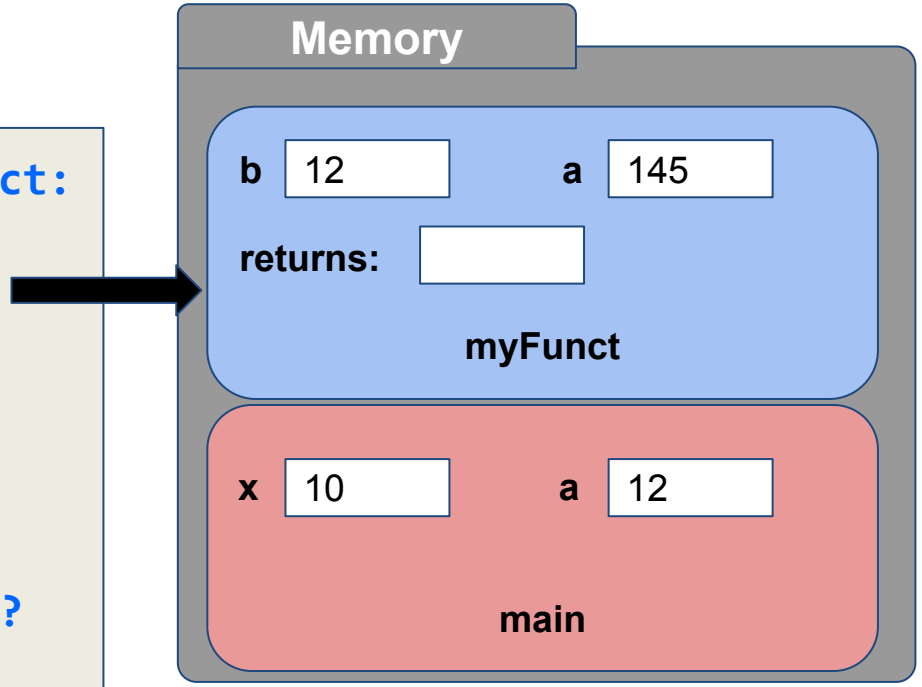


A stack is created in memory, in which the function's local variables are stored

4.1. Functions and their parameters: Using Functions

- A function is *called*:

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

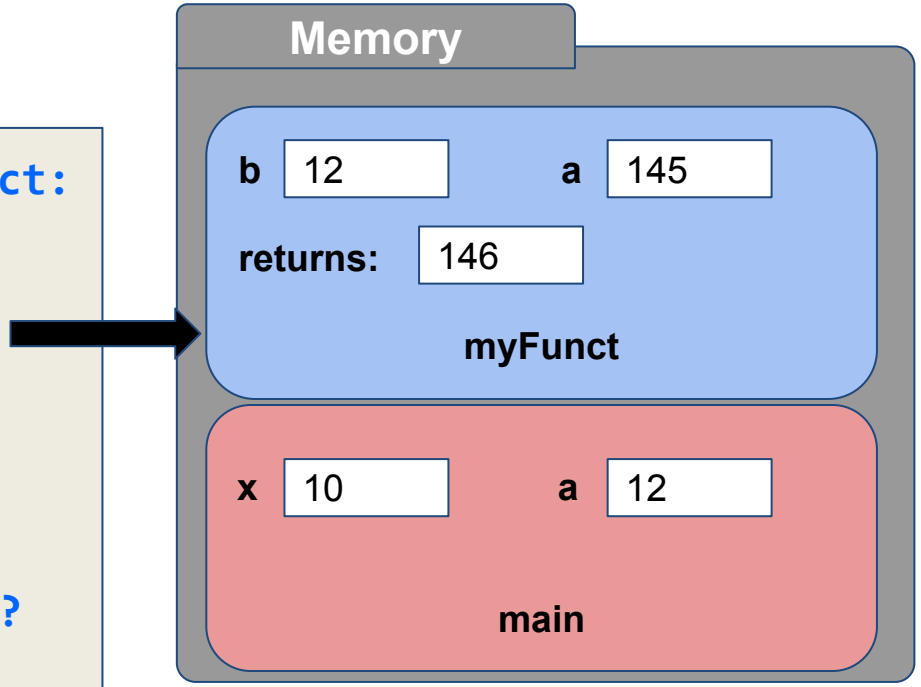


A stack is created in memory, in which the function's local variables are stored

4.1. Functions and their parameters: Using Functions

- A function is *called*:

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

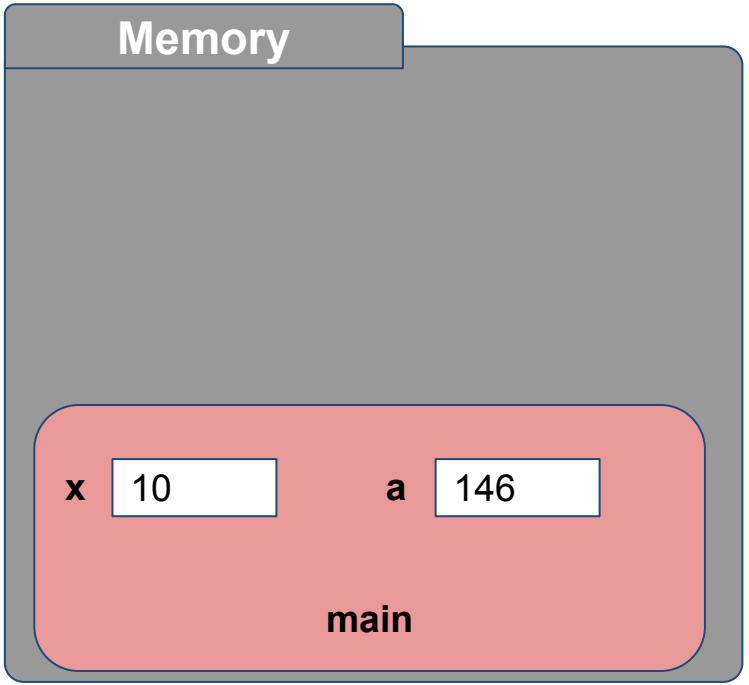


A stack is created in memory, in which the function's local variables are stored

4.1. Functions and their parameters: Using Functions

- A function is *called*:

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



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

4.1. Functions and their parameters: Using Functions

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

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

4.1. Functions and their parameters: Using Functions

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

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

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

4.1. Functions and their parameters: Using Functions

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

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

4.1. Functions and their parameters: Using Functions

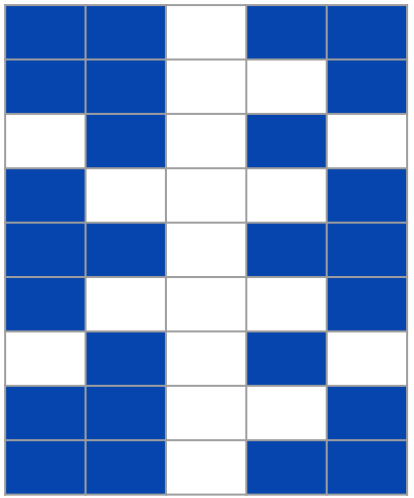
```
int main() {
  auto c = ' ';          // used for user key input
  auto x = 10, y = 10;   // (x,y) position of player: start at (10,10)
  initNCurses();         // initialize ncurses functionality
  while ( c != 'q' ) {    // as long as the user doesn't press q ..
    clearScreen();
    draw(x, y,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

4.1. 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.

```
#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:

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

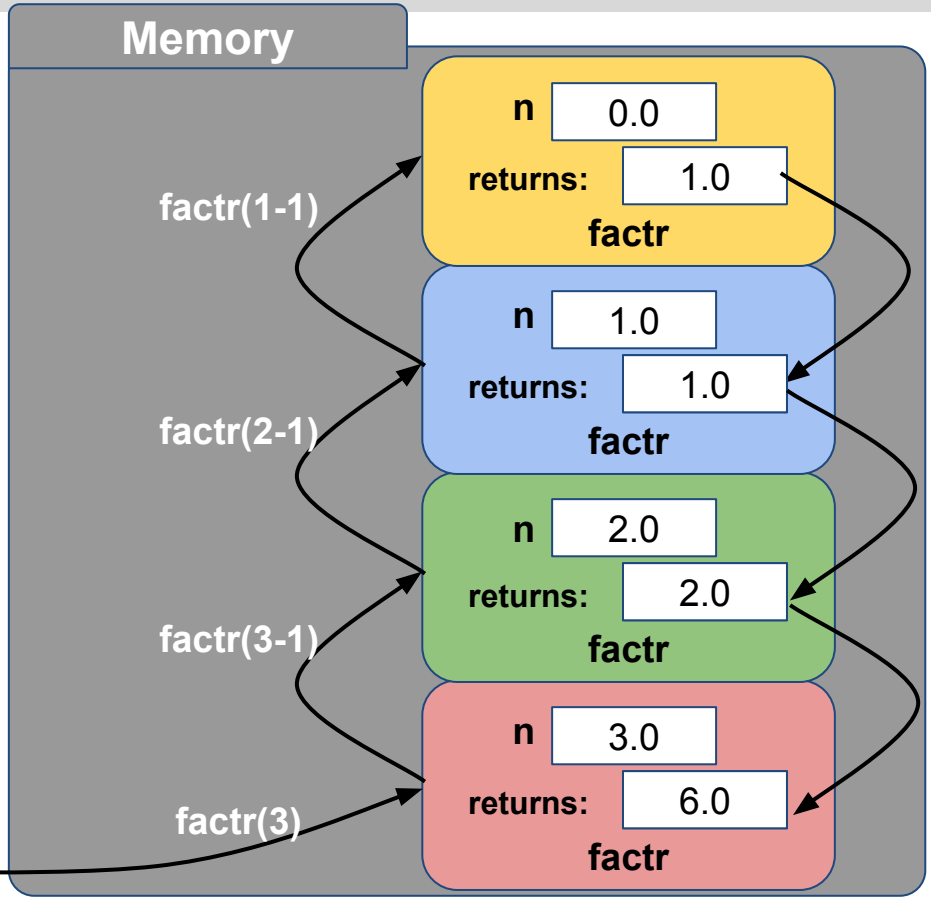
so:
2! = 2 . 1 = 2
3! = 3 . 2 . 1 = 6
4! = 4 . 3 . 2 . 1 = 24
5! = 5 . 4 . 3 . 2 . 1 = 120
and so on …

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);
```



4.3. Call by Value

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

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

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

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

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

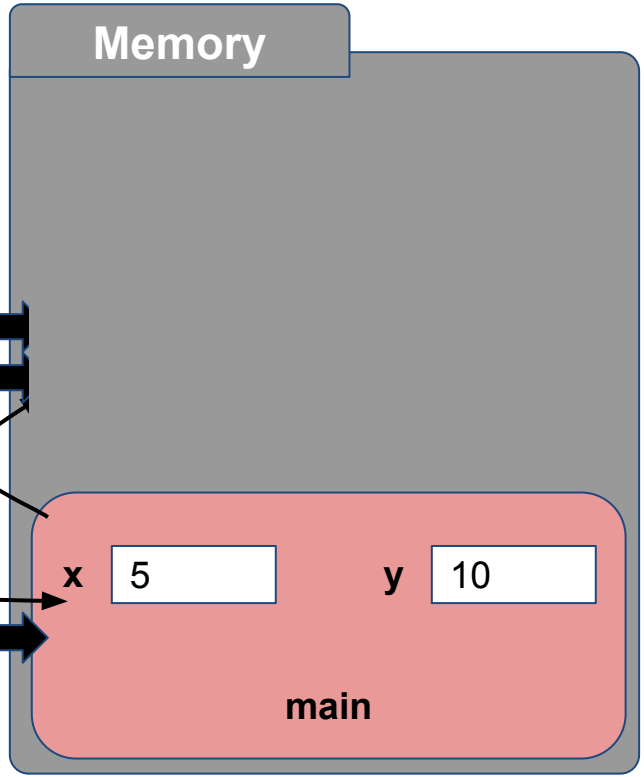
```
int a = 10;
y = maximum(a, a);    // the value 10 is copied to both parameters
```

4.3. Call by Value

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

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

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

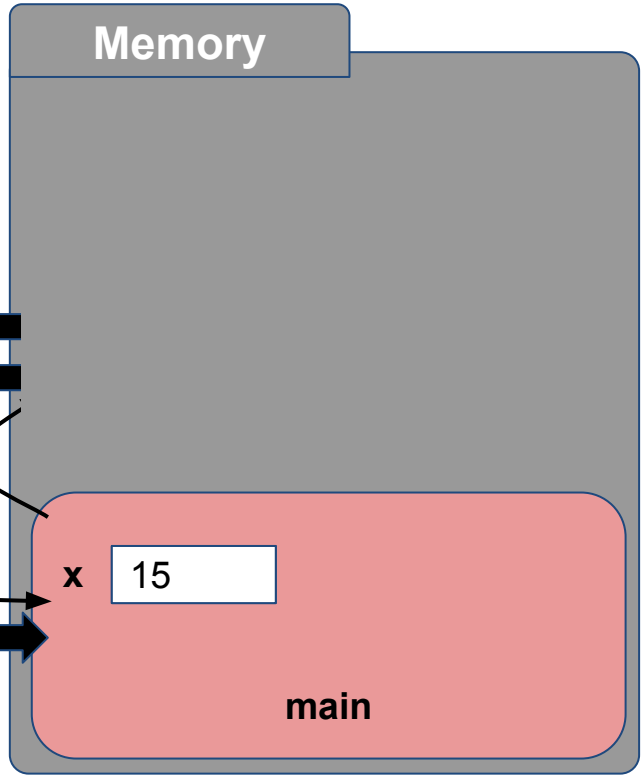


4.3. Call by Value

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 << '\n';
  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 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

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

4.6. Header files and Modules

drawMaze.h

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

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

// clear the screen
void clearScreen();

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

4.6. Header files and Modules

drawMaze.cpp

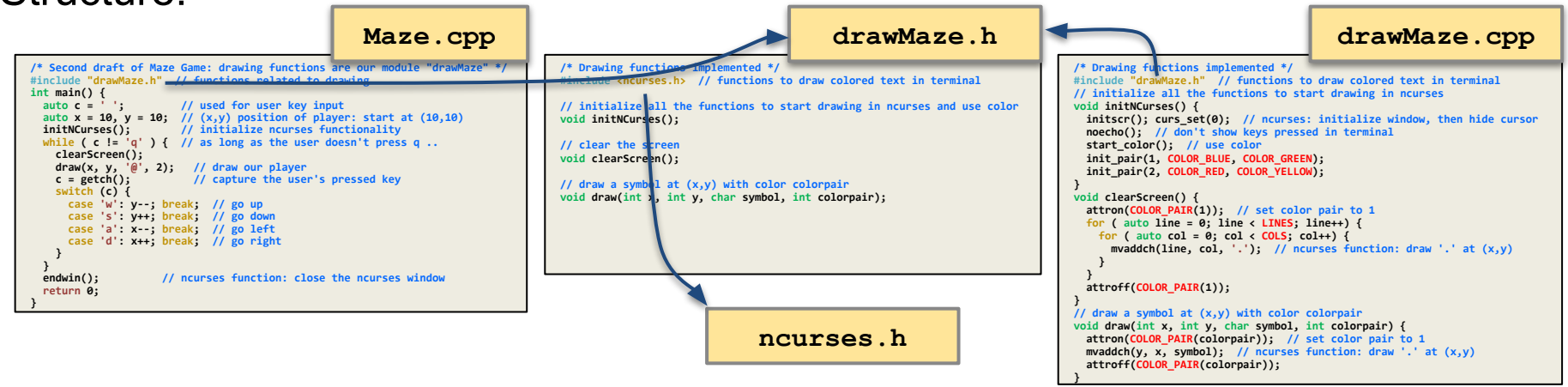
```
/* Drawing functions implemented */
#include "drawMaze.h"  // functions to draw colored text in terminal

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

4.6. Header files and Modules

drawMaze.cpp

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

Structure:



4.6. Header files and Modules

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

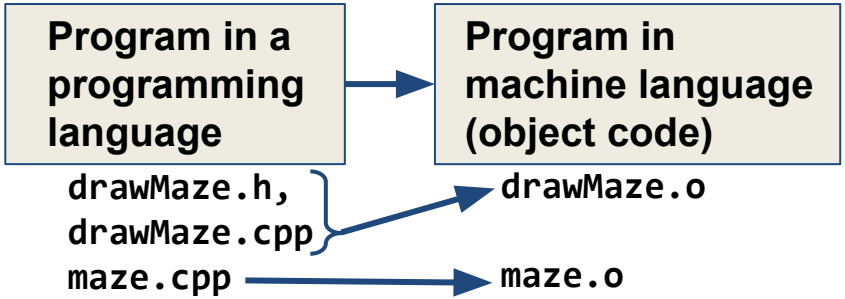
- First compile the module and the program into object files:

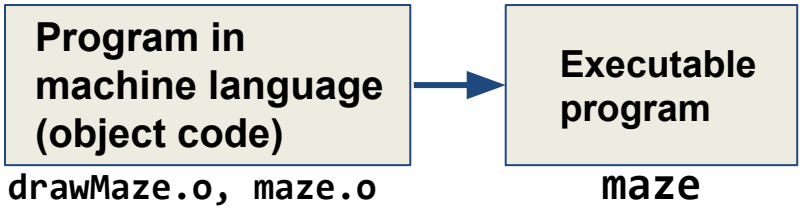
  `g++ -c drawMaze.cpp -std=c++11` → object file `drawMaze.o`

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

- Then link the object files:

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



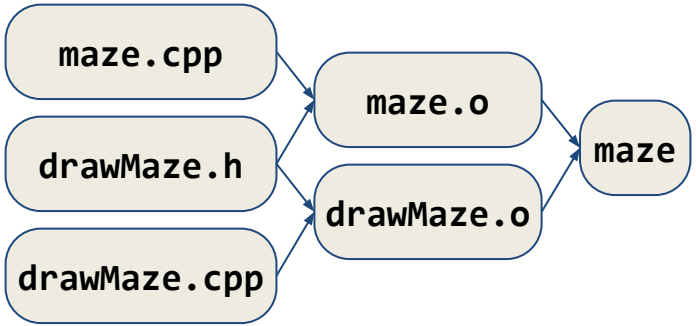


4.6. Header files and Modules

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

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

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



compile **drawMaze.cpp:**

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

compile **maze.cpp:**

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

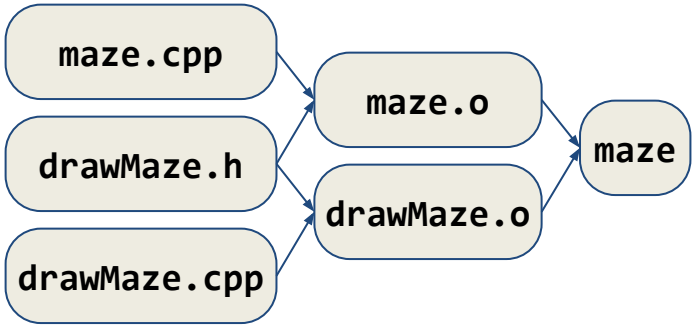
link the objects files into the executable program **maze:**

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

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

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

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



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

Makefile

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

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)

```
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);
}
```

```
#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;
  }
}
```