INTRODUCTION TO C++
BEFORE OBJECTS

Trainer: Michał Rad

Author: Bartłomiej Kozak

AGENDA

- **Introduction**
  - Program structure
  - Compilation process
  - Preprocessor directives
- **C++ Language Basics**
  - Variables
  - Pointers and References
  - Namespaces
- **Program Flow Control**
  - Input / Output streams
  - Branching and Looping

AGENDA

- **Functions**
  - Passing arguments by value / reference
  - Name overloading
  - Default argument values
  - Reading complex declarations
- **Compound Data Types**
  - Arrays
  - Structures and Unions
- **Dynamic Memory Allocation**
  - Operators `new` and `delete`

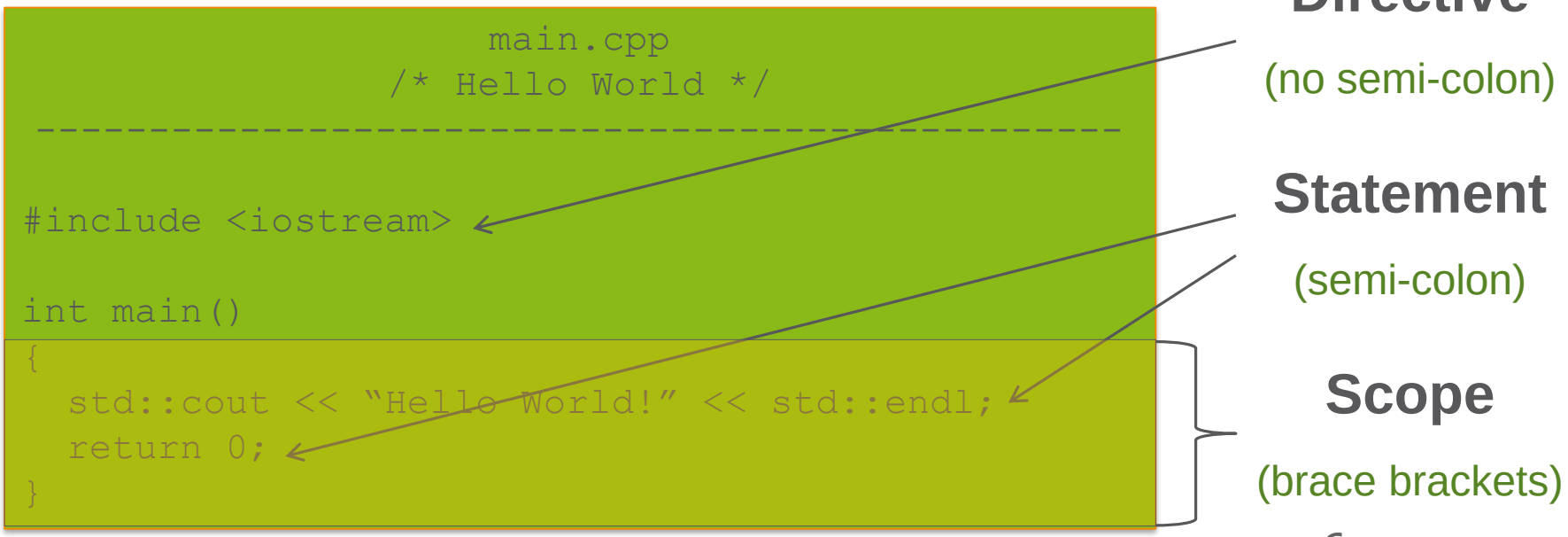
Programming is like chess.

Proficiency comes with practise!

CHAPTER I
INTRODUCTION TO C++

C++ code structure:

- Program is composed of directives and statements
- Every statement is terminated by semi-colon (;)
- Statements are collected into sections (scopes)



Code style in a single file:

- Line Breaks – one statement per line
- Blank Lines – offset the components of code
  - After precompiler declarations
  - After new variables are declared
- Indentation – a new block of code should be indented by one tab more than the code in the previous path

Those are only suggestions!

Most of the project have their own style.

Comments are used to:

› Explain specific parts of the code

› Describe the properties or show the info about the file

```
// Single line comment (preferable)
```

```
/* Multi-line comment
   (can not overlap) */
```

Comment WHY not HOW!

Program in C++ is organized in two file types:

- Header files (often `*.h`, `*.hpp` or `*.hxx`)
- Source files (often `*.c`, `*.cpp` or `*.cxx`)

functions.h

```
void displayHello();
```

Header

(declarations)

What can I do?

functions.cpp

```
#include <iostream>
#include "functions.h"

void displayHello()
{
  std::cout << "Hello!";
}
```

Source

(definitions)

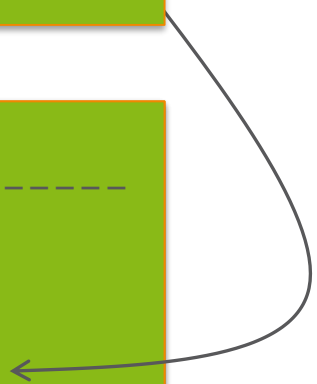
How will I do it?

Connecting files:

› Preprocessor directive `#include`

```
        functions.h
 -------------------------
int i;
void displayHello();
```

```
        functions.cpp
 -------------------------

#include <iostream>
int i;
void displayHello();

void displayHello()
{
  std::cout << "Hello!";
}
```



Preprocessor replaces the `#include` directive with the text of the specified file.

`#include <file>` – file is searched in the compiler include paths

`#include "file"` – search is expanded to include the current source directory

C++ program starting point:

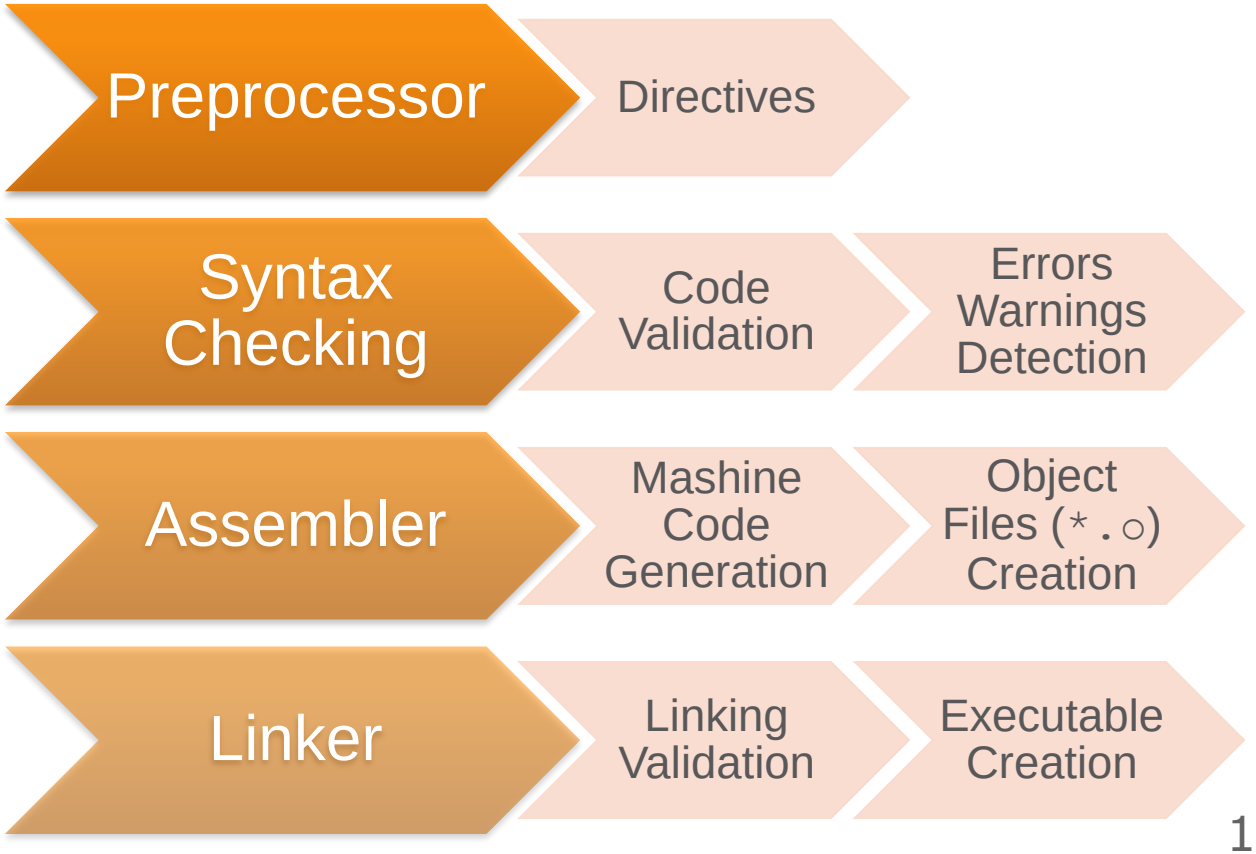
- The starting point of every C++ program is a function named `main()`
- Every program must have exactly one such function
- Function `main()` declaration may vary on different compilers

```
                    main.cpp
                /* Hello World */
 ----------------------------------------------

#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Starting Point

(function `main()`)

Compilation – process of transforming a human readable code into machine code comprehensible to a microprocessor.



Compilation process – Preprocessor

- The preprocessor is not a part of the C++
- All preprocessor directives begin with the hash character (#)

Inclusion of Files

Macro Expansions

Conditional Compilation

Other…

Macro definitions and expansion (`#define` and `#undef`) directives allows for the defining constants and function-like macros.

```
#define <identifier> <replacement>
#undef <identifier>
```

```
#define PI 3.14

double circleArea(double r)
{
  return PI * r * r;
}

#undef PI
```

Preprocesor

```
#define PI 3.14

double circleArea(double r)
{
  return 3.14 * r * r;
}

#undef PI
```

Macro definitions - function-like macros:

- Simple replacement mechanism

```
#define <identifier><(parameters)> <replacement>
```

```
#define SQUARE(x) x*x
```

```
double A = SQUARE(2);
double B = SQUARE(1+2);
```

Preprocesor

```
double A = 2*2;
double B = 1+2*1+2;
```

4

5

```
#define SQUARE(x) ((x)*(x))
```

```
double A = SQUARE(2);
double B = SQUARE(1+2);
```

Preprocesor

```
double A = ((2)*(2));
double B = ((1+2)*(1+2));
```

4

9

Be very careful when using function-like macros.

Conditional inclusions (`#ifdef`, `#ifndef`, `#if`, `#endif`, `#else` and `#elif`) directives allows for the specified parts of the code to be compiled under special condition.

```
#if VERSION > 1
  std::cout << "Ver. is higher than 1.0" << std::endl;
#endif
```

```
#ifdef _WIN32
  std::cout << "System is Windows" << std::endl;
  #include <windows>
#elif defined _unix__
  std::cout << "System is Unix" << std::endl;
  #include <unistd>
#endif
```

Compilation process – Syntax Checking

- Code validation
- Errors and warnings detection (file name and line number can be provided)

```
int i = 5;
int k = j + 2;
```

Undefined variable ‘j’

```
int i = 4;
int k = i + 3
```

Missing ‘;’ at the end of line

```
int i = 0;
fur (i = 1; i < 2; ++i)
{
}
```

Implicit declaration of function ‘fur’

Expected ‘)’ before ‘;’

...

Compilation process – Object Code

- Machine code generation
- Object files (.o) creation
- The code itself can't be executed at this stage

*.cpp

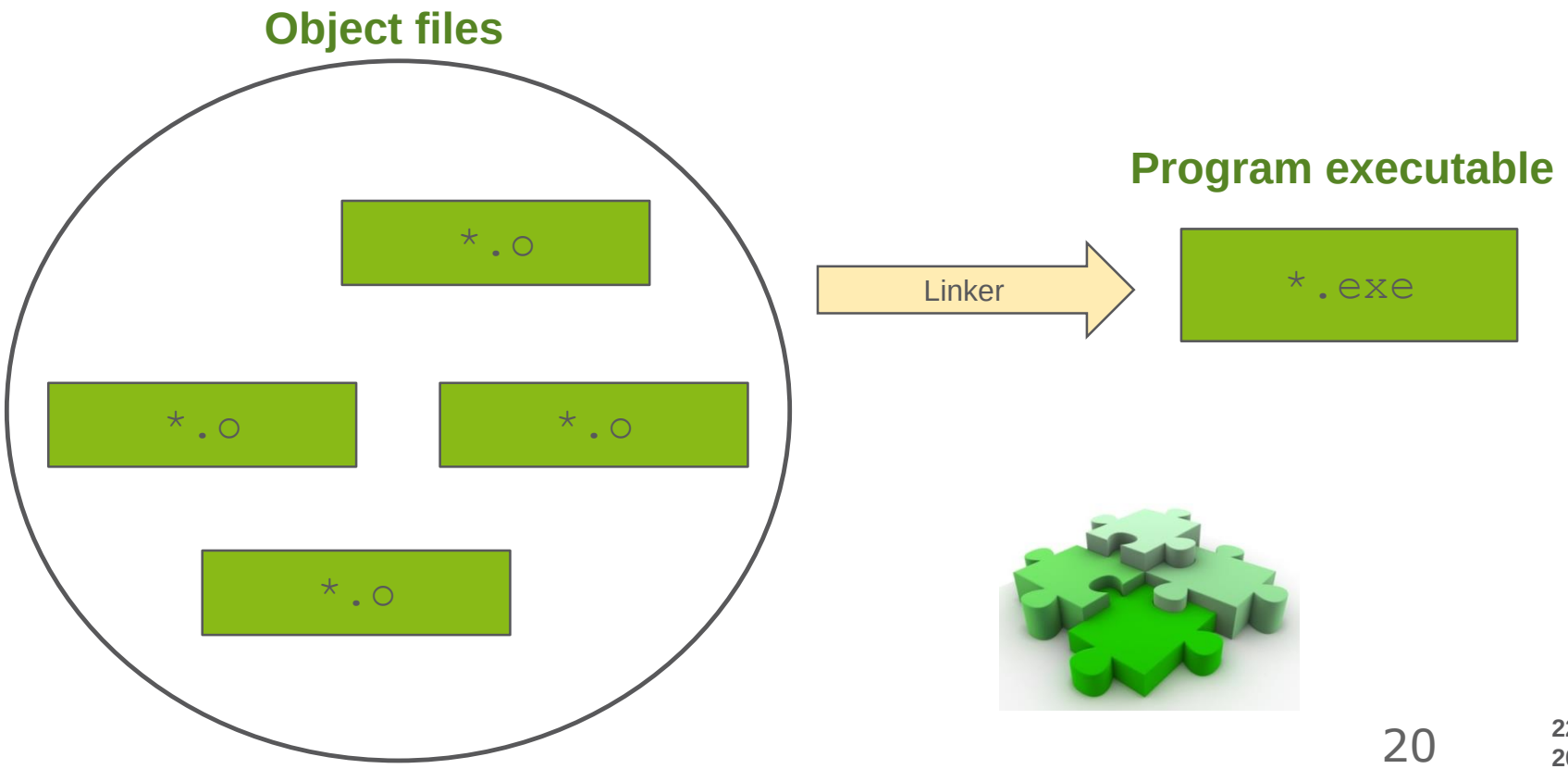
```
int main()
{
    int a = 5;
    return a;
}
```

*.o

```
01000101010100111
01010100110100101
00010011100011001
01010000111011010
10111010100100101
```

Compilation process – Linking

- Linking validation
- Performed by a linker



Summary:

- **Program structure**
  - Header files
  - Source files
  - `main()`
- **Compilation process**
  - Preprocessor directives
    - `#include`, `#define` - macro
  - Syntax checking
  - Object code
  - Linking

CHAPTER II
C++ LANGUAGE BASICS

Identifier is a name of a resource (variable, function, constant, structure etc...).

Allowed characters:

- Letters (upper and lower case)
- Digits
- The underscore (_) and the dollar ($) characters
- May not begin with a digit
- There are 63 reserved words in C++

`foo`	✓	`ELEMENT$`	✓
`3$Doors`	✗	`No-time`	✗
`_foo_2`	✓	`switch`	✗

Name	Description	Size	Range
`char`	Character or small integer	1 byte	`-128 to 127`
`int`	Integer	4 bytes	`-2147483648 to 2147483647`
`float`	Floating point number	4 bytes	`+/- 3.4e +/- 38 (~7 digits)`
`double`	Double precision floating point number	8 bytes	`+/- 1.7e +/- 38 (~15 digits)`
`bool`	Boolean value True or False	1 byte	`true / false`

Operator	Syntax
Assigment	a = b
Addition	a + b
Substration	a – b
Multiplication	a * b
Division	a / b
Modulo	a % b
Increment	a++ or ++a
Decrement	a-- or --a

Operator	Syntax
Addition assignment	a += b
Subtraction assignment	a -= b
Multiplication assignment	a *= b
Division assignment	a /= b
Modulo assignment	a %= b

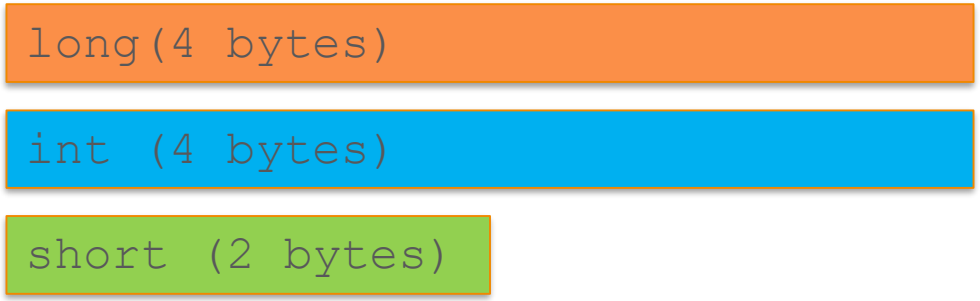
Operator	Syntax
Bitwise NOT	~a
Bitwise AND	a & b
Bitwise OR	a | b
Bitwise XOR	a ^ b
Bitwise left shift	a << b
Bitwise right shift	a >> b

Operator	Syntax
Bitwise AND assignment	a &= b
Bitwise OR assignment	a |= b
Bitwise XOR assignment	a ^= b
Bitwise left shift assignment	a <<= b
Bitwise right shift assignment	a >>= b

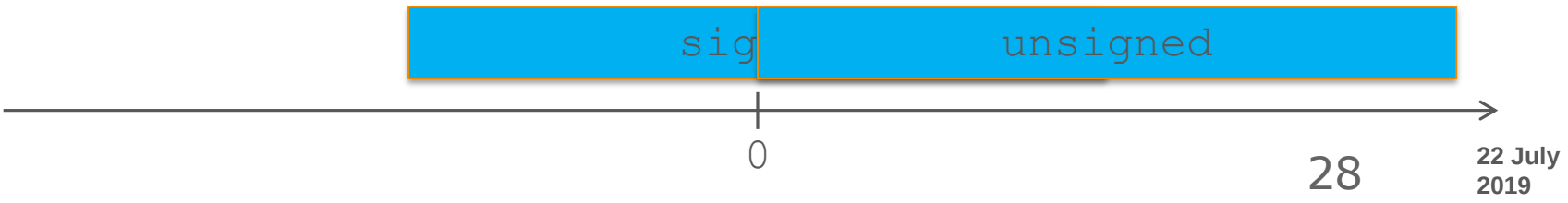
Operator	Syntax
Equal to	a == b
Not equal to	a != b
Greater than	a > b
Less than	a < b
Greater than or equal	a >= b
Less than or equal	a <= b
Logical NOT	!a
Logical AND	a && b
Logical OR	a || b

Operator	Syntax
Array subscript	a[b]
Indirection	*a
Reference	&a
Structure dereference	a->b
Structure reference	a.b
Function call	a(a1, b2)
Comma	a, b
Ternary conditional	a ? b : c
Size-of	sizeof(type)

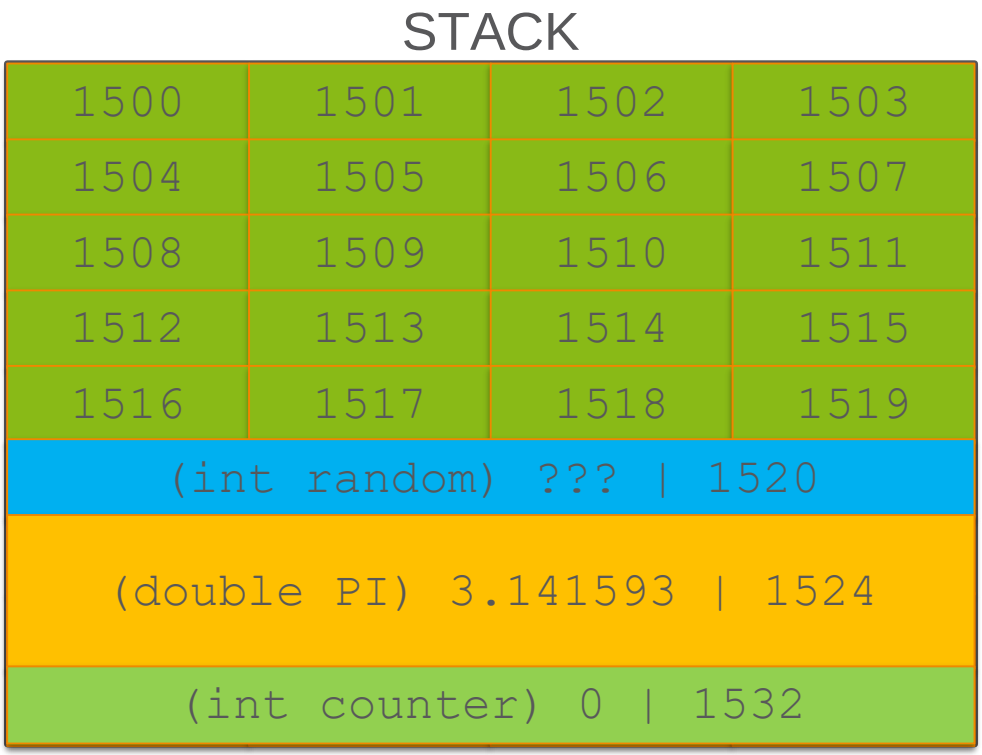
› `short / long` – provide different lengths of integers.



- `signed (default) / unsigned` – may be applied to any integer type. Unsigned variables are always positive or zero



Variable is a building block of all algorithms – portion of memory that holds a value.



Stack

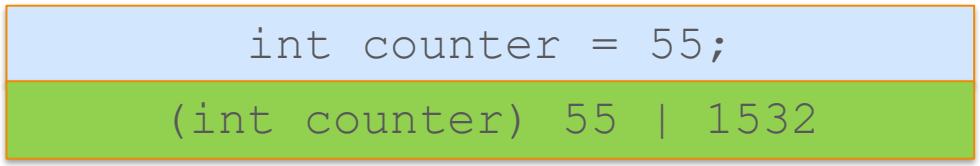
Memory area

Multiple cells, each with its own address

```
int counter = 0;
```

```
double PI = 3.141593;
```

```
int random;
```



Data Type

(int)

Name

(counter)

Value

(55)

Address

(1532)

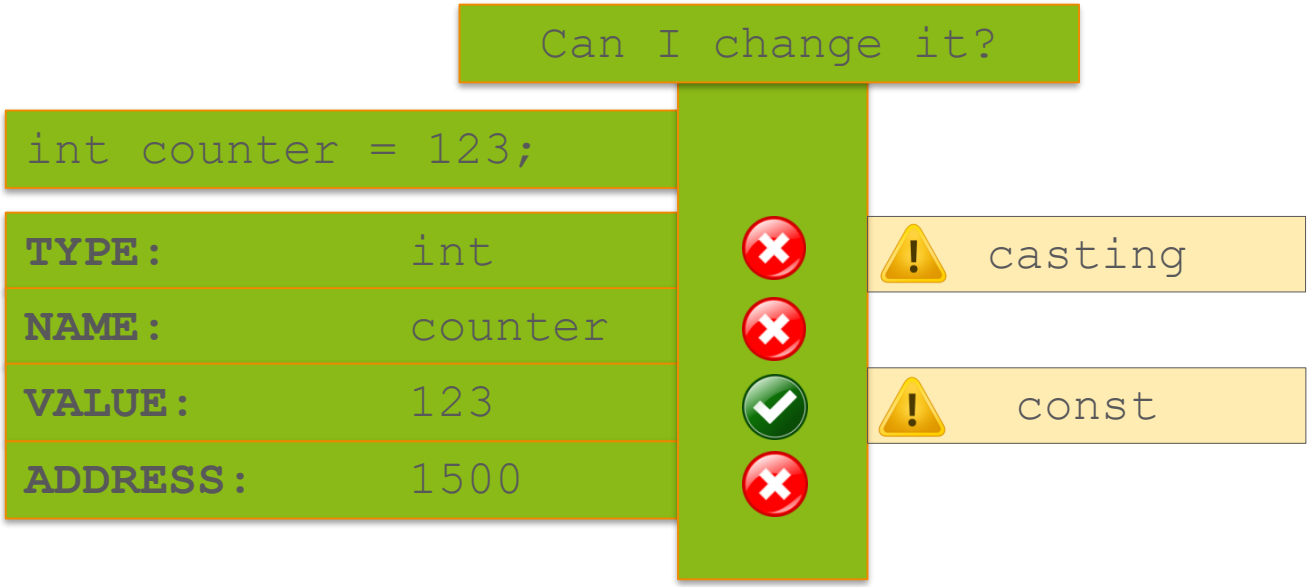
ACCESS:

counter **55**

&counter **1532**

Variable consists of the following:

- Type
- Identifier (name)
- Value
- Address



Constants are expressions with a fixed value.

Literals (explicit constant):

- Used to give concrete values to variables

```
int a = 5;
float b = 3.54;
char c = 'i';
```

```
75         // decimal
0113       // octal
0x4b       // hexadecimal
0b1001011 // binary
        (gcc or C++14)
```

Declared constants (`const`):

- Declared using `const` prefix
- Value must be given during declaration

```
const int width = 800;
const int height = 600;
const float PI = 3.14;
```

Casting is a mechanism of temporary data type change.

```
100 + 'a' = ?
```

1. Type of the result?

2. Value of the result?

ASCI code for `'a'` is 97. Hence the result is `int` with value of 197.

Automatic (implicit):

› Variable of a shorter data type transformed to a variable with longer data type.

```
short a = 5;  // 2 bytes
int b;        // 4 bytes
b = a;        // short -> int (2 bytes -> 4 bytes)
```

Given (explicit):

- User may explicit show the data type to cast to
- Higher priority than automatic casting

```
short a = 5;
int b;
b = (int)a; // C-like cast notation (old)
b = int(a); // function like C++ notation (new)
```

Pointer is a variable that contains the address of a memory cell.

```
int i = 12;
```

```
int *p = &i;
```

STACK

1500	1501	1502	1503
1504	1505	1506	1507
1508	1509	1510	1511
1512	1513	1514	1515
(int *p) 1520 | 1516			
(int i) 12 | 1520			

Pointer

Points to a given variable type.

Value of a pointer is an address.

Declaration of a pointer:

```
<data type> *<pointer name>;
```

In order to access the variable stored in the address indicated by pointer, dereference (*) operator must be used.

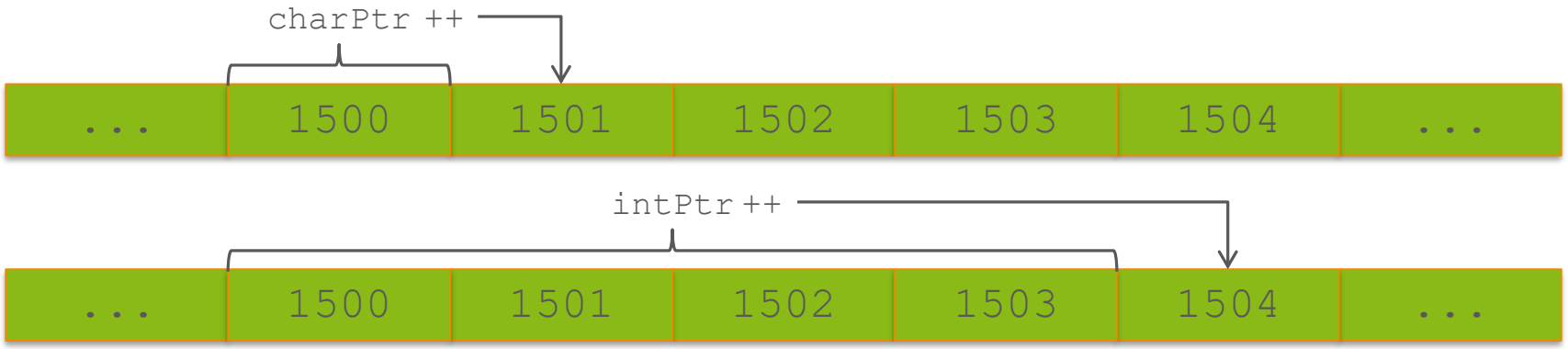
```
int i = 5;
int *ptr;

ptr = &i;

std::cout << "Address of i: " << ptr;
std::cout << "Value of i: " << *ptr;
```

```
Address of i: 1500
Value of i: 5
```

Arithmetic of pointers is different than the one performed on regular data types:

- Only addition and subtraction is allowed
- Its behavior changes with different data types

```
char *charPtr;
int *intPtr;

charPtr++;
intPtr++;
```



Special type of pointer is pointer to void (void*):

- Represents the “absence” of type
- Length of variable is undefined – can be assigned to point any data type
- Dereference properties are undetermined – can’t be dereferenced directly (cast must be used)

```
char letter = ‘a’;
int number = 25;
void *pointer;

pointer = &letter;
std::cout << “Letter: ” << *((char*)(pointer));

pointer = &number;
std::cout << “Number: ” << *((int*)(pointer));
```

Casting

(to `char*`)

Dereference

(reach for value)

Special value for pointer is NULL:

- Regular pointer
- Indicates that the pointer is not pointing to any valid reference or memory address
- Represented by integer zero or by macro NULL (preferred use)

```
int *p = NULL;
```

Using NULL protects programmer from accessing memory the pointer would point otherwise.

NULL is often a returned value by functions (returning pointers) that indicates that error occurred.

Reference is an alias (another name) for existing variable:

- Must be initialized when declared.

```
int i = 12;
```

```
int &r = i;
```

STACK

1500	1501	1502	1503
1504	1505	1506	1507
1508	1509	1510	1511
1512	1513	1514	1515
1516	1517	1518	1519
(int i, r) 12| 1520			

Reference

Must be of the same type.

Now you can:

Use (`r`) instead of (`i`)

No additional memory is used

Alias for the data type can be created (`typedef`).

Variable that is 8 bits long, with only positive numbers:

(`unsigned char`)

u8

Variable that is 16 bits long, with only positive numbers:

(`unsigned short`)

u16

Variable that is 32 bits long, with only positive numbers:

(`unsigned int`)

u32

```
typedef <data type> <alias>;
```

```
typedef unsigned char      u8;
typedef unsigned short     u16;
typedef unsigned int       u32;
```

Which is better?

```
int day = 1;
if (day > 5) {
  std::cout << "weekend";
}
```

```
DAY day = Monday;
if (day > Friday) {
  std::cout << "weekend";
}
```

Enumeration is a set of named integer constants.

```
enum <name>
{
  <tag #0>,   // 0
  <tag #1>,   // 1
  <tag #2>,   // 2
  …,          // …
  <tag #n>    // n
};
```

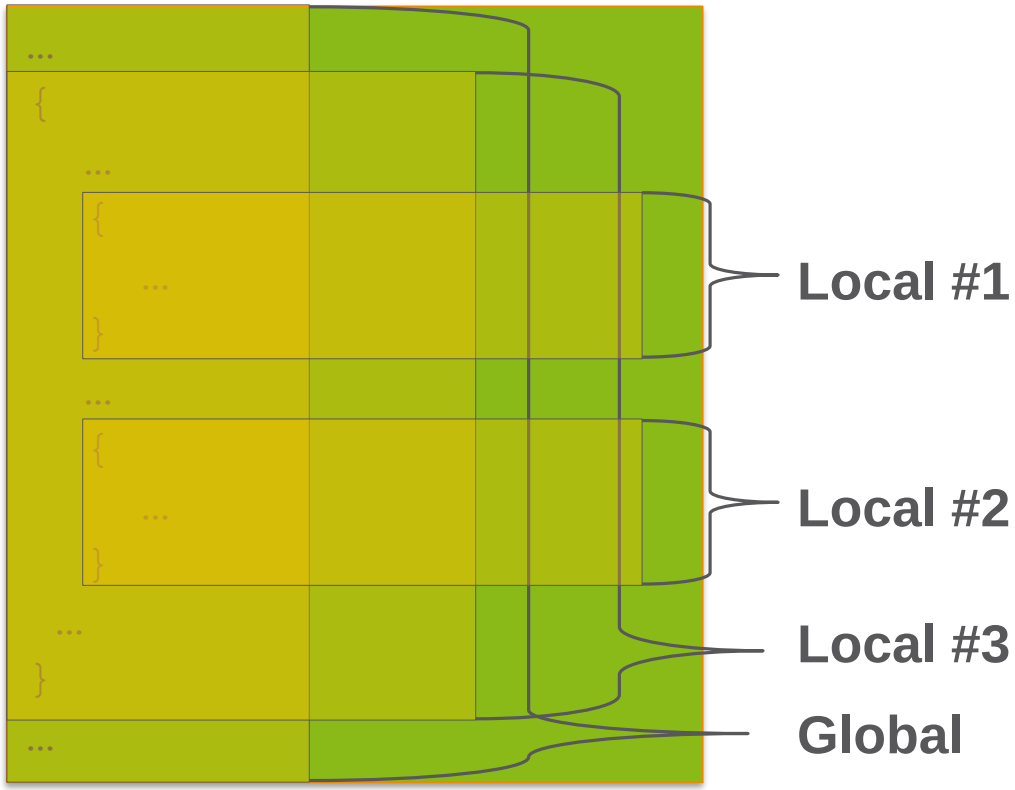
```
enum <name>
{
  <tag #0> = <value #0>,
  <tag #1> = <value #1>,
  <tag #2> = <value #2>,
  …,          // …
  <tag #n> = <value #n>
};
```

- Enumeration variables are still integers
- Values can be set by hand
- Value increases by 1 if not specified

```
enum DAY
{
  Monday,     // 0
  Tuesday,    // 1
  Wednesday,  // 2
  Thursday,   // 3
  Friday,     // etc…
  Saturday,
  Sunday
};

DAY currentDay = Wednesday;
```

Scope is used to split code into smaller parts:

- Determined by { } (braces)
- Resources can be referenced only in their scope
- Helps organize the program

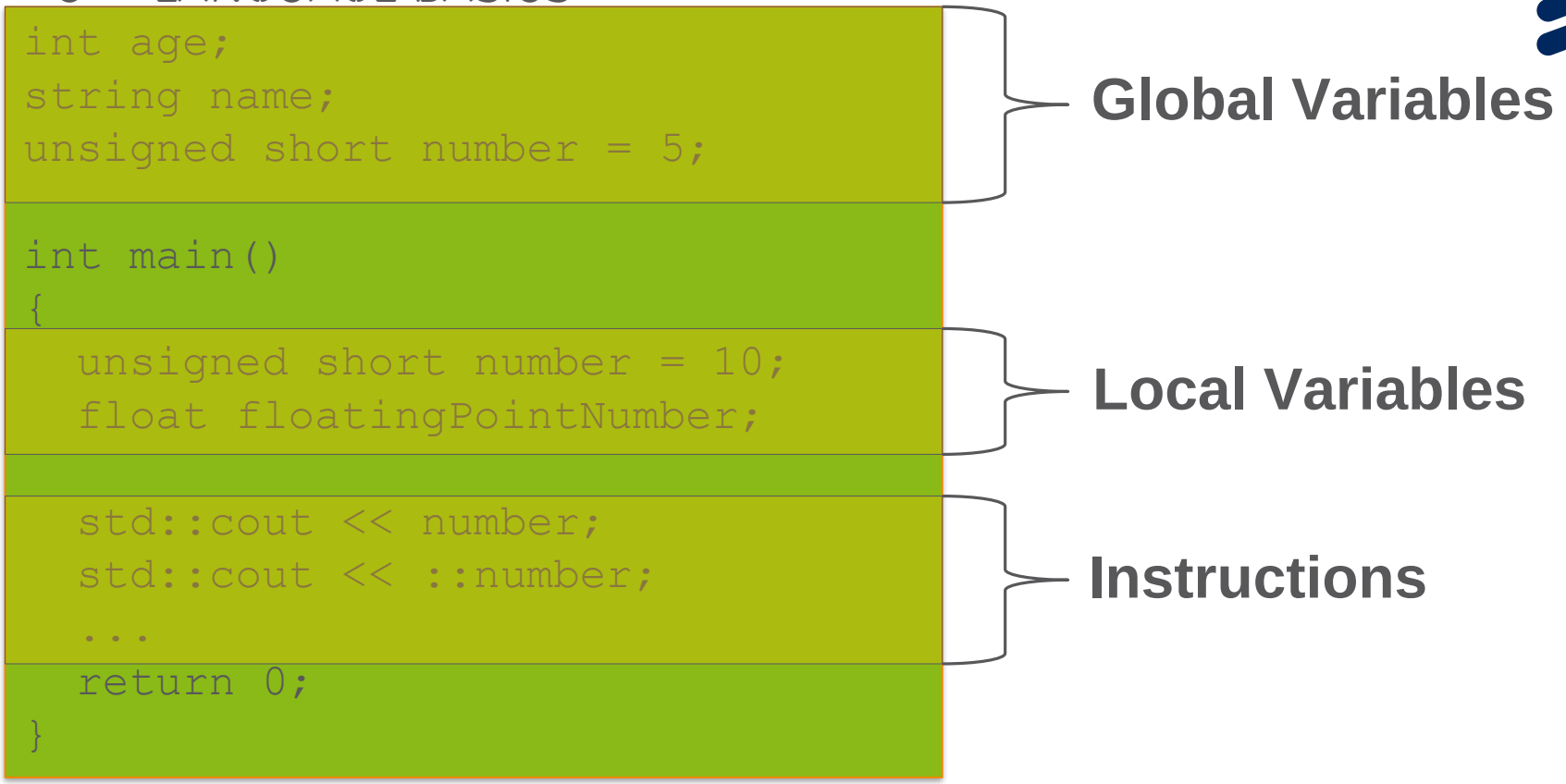


Global resources:

- Visible in the entire code after being defined
- Variables initialized as 0 by default
- Destroyed when the program ends

Local resources:

- Visible only in the block level in which it is defined
- Not initialized
- Destroyed after leaving the block



- Local resource takes preference over global
- Global resource may be accessed using scope (`::`) operator

Scope determines where the variable can be referenced to.

Global variable:

- Visible in the entire code after being declared
- Initialized as 0 by default

Local variable:

- Should be declared at the beginning of a block
- Visible only in the block level in which it is declared
- Destroyed after leaving the block
- Not initialized

Namespaces allow to group resources under a given name:

```
namespace <namespace name>
{
  // namespace entities
}
```

```
namespace myNamespace
{
  int a, b, c;
}
```

Namespaces allows to:

- Divide global scope into “sub-scopes”, each with its own name
- Better organize the code

In order to access a variable in a namespace, from outside the namespace, scope operator (`::`) must be used.

```
int a = 5;

namespace myNamespace {
  int a = 10;
}

int main() {
  int a = 15;

  std::cout << "Global: " << ::a;
  std::cout << "From namespace: " << myNamespace::a;
  std::cout << "Local: " << a;

  return 0;
}
```

```
#include <iostream>

int main() {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Instead of using full namespace path, `using` directive can be used.

```
#include <iostream>

using std::cout;
using std::endl;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

Whole namespace can be used:

```
using namespace <namespace name>;
```

For example:

```
using namespace std;
```

Summary:

- **Variables**
  - Type, name, value, address
  - Pointer
  - Reference
  - Operators
  - Typedef, enumeration type
- **Scope**
  - Local
  - Global
- **Namespaces**
  - `std`

CHAPTER III
PROGRAM FLOW CONTROL

String is an object that represent sequence of characters.

- Standard string library (`#include <string>`) must be included (`namespace std`)
- Very intuitive
- Full documentation: http://www.cplusplus.com/reference/string/string/

```
#include <string>
using namespace std;

string name = "John";                    // assignment
string fullName = name + " Smith";       // append
if (fullName == "John Smith")…           // comparison
int len = fullName.length();             // length of string
```

Input / Output operations in C++ are performed using a convenient abstraction called streams.

- Characters can be inserted or extracted from a stream
- Standard C++ library header `iostream` is required in order to use the standard input / output stream (`#include <iostream>`)
- By default standard input is the keyboard
- By default standard (also standard error) output is the screen / console

C++ stream object defined to access standard input is `cin`.

- Reading data from `cin` stream is performed using (`>>`) (shift right) operator
- More than one `>>` operator can be used in a single statement
- Any build in types can be read from input stream
- Can be used with manipulators

```
cin >> <variable or manipulator>;
```

```
int i;
float f;
cin >> i >> f;      // separate i and f with a whitespace
```

Manipulator	Description
boolalpha	Alphanumerical bool values
noboolalpha	No alphanumerical bool values
skipws	Skip whitespaces
noskipws	Do not skip whitespaces
dec	Use decimal base
hex	Use hexadecimal base
oct	Use octal base
ws	Extract whitespaces

- Whitespace character stops extraction
- Entire line can be extracted using the `getline` function (`iostream` library)

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

using namespace std;

int main ()
{
  string name;
  cout << "What's your name? ";
  getline (cin, name);
  cout << "Hello " << name << endl;
  return 0;
}
```

Input Stream

Where to put?

C++ stream object defined to access standard output is `cout`.

- In order to insert data into `cout` stream (`<<`) (shift left) operator must be used
- More than one `<<` operator can be used in a single statement
- Any build in types can be put into stream
- Manipulators can change the display format

```
cout << <value / variable or manipulator>;
```

```
cout << "Hello World!";     // Hello World!
cout << 255;                // 255
cout << hex << 10;          // a (10 in hex is a)
cout << 10 << " c";         // a c (manipulator is still on)
```

Manipulator	Description
dec	Use decimal base
endl	Insert newline and flush
flush	Flush stream buffer
hex	Use hexadecimal base
scientific	Use scientific floating-point notation
showpos	Show positive signs
uppercase	Generate upper-case letters

Manipulators:

- Single case use
- Permanent – works for the stream until changed

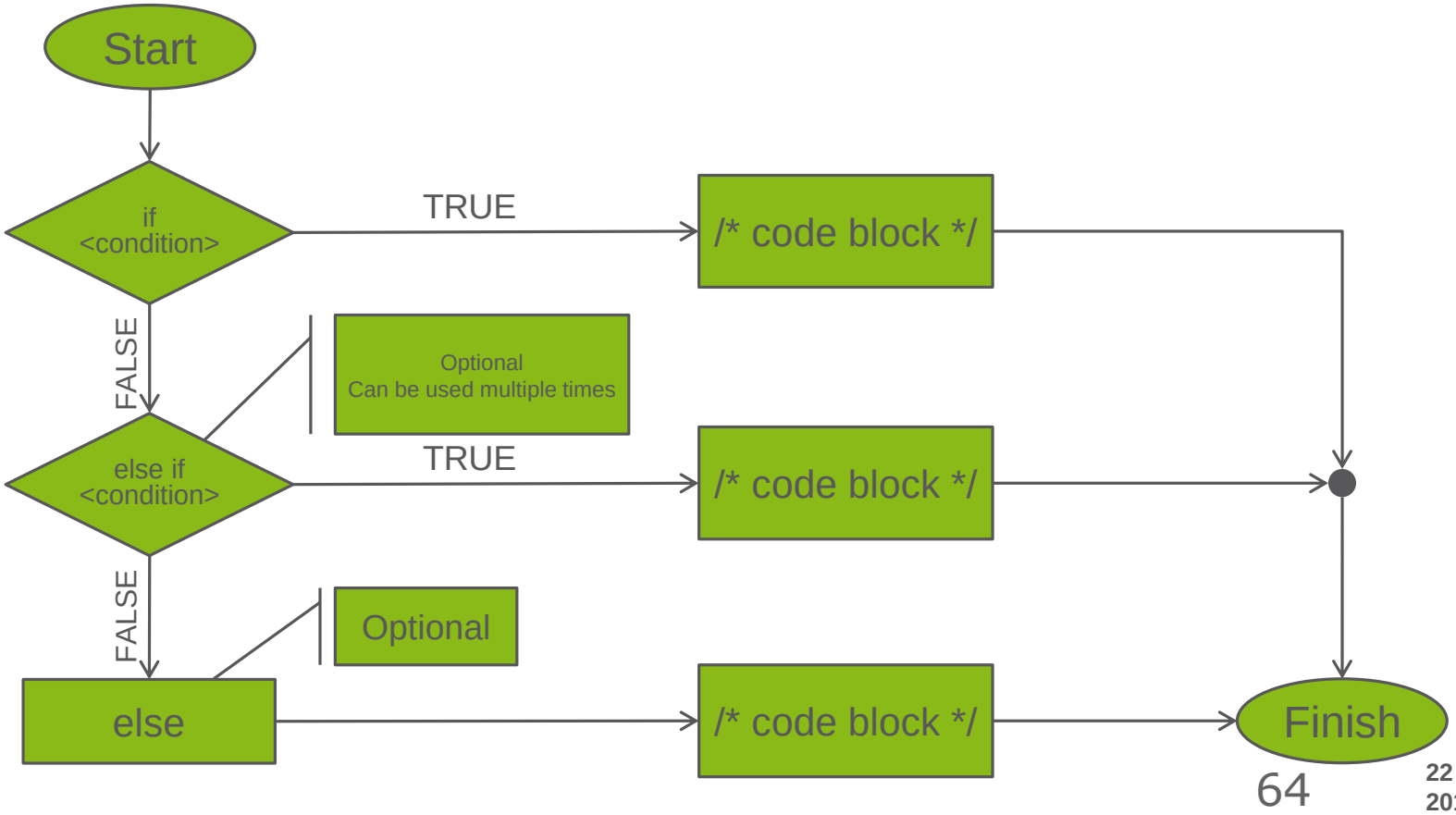
C++ program is not limited to linear sequence of instructions. During its process it may:

- **Repeat code** – fragment of code is executed number of times specified by the programmer
- **Take decisions** – program can make decisions during runtime based on the current conditions
- **Reuse code** – some parts of the program may be reused in different places

Programming Language C++ provides two styles of flow control:

- **Branching** – program must choose to follow one branch or another
  - If-Else
  - Conditional expression
  - Switch-Case
- **Looping** – way to repeat commands and control how many times they are repeated
  - While
  - Do-While
  - For

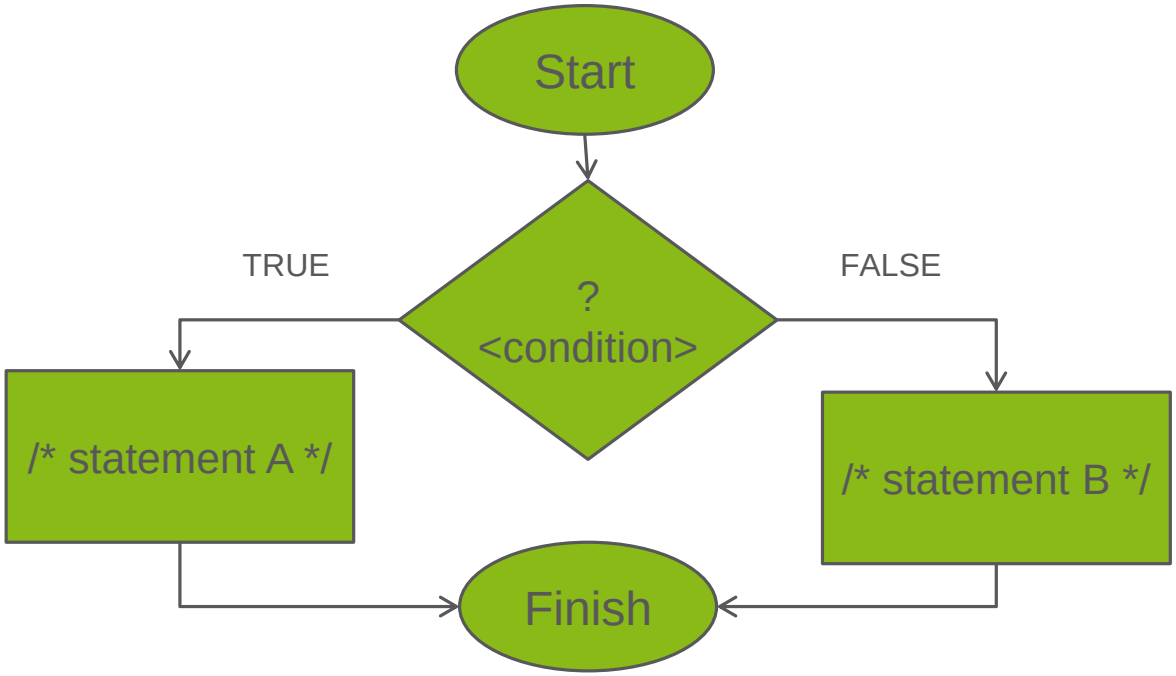
If-Else statement provides a way to instruct the computer to execute a block of code only if certain condition is met.



If-Else statement:

- Only one `if` expression can be used and it is obligatory
- Multiple `else if` and only one `else` expression can be used but there are optional
- Only one code block can be executed

Conditional Operator (`? :`) is a ternary (takes three arguments) operator that checks a single logical expression.



Condition

Implicitly converted to bool.

Statement #1

Evaluated if condition is true.

Statement #2

Evaluated if condition is false.

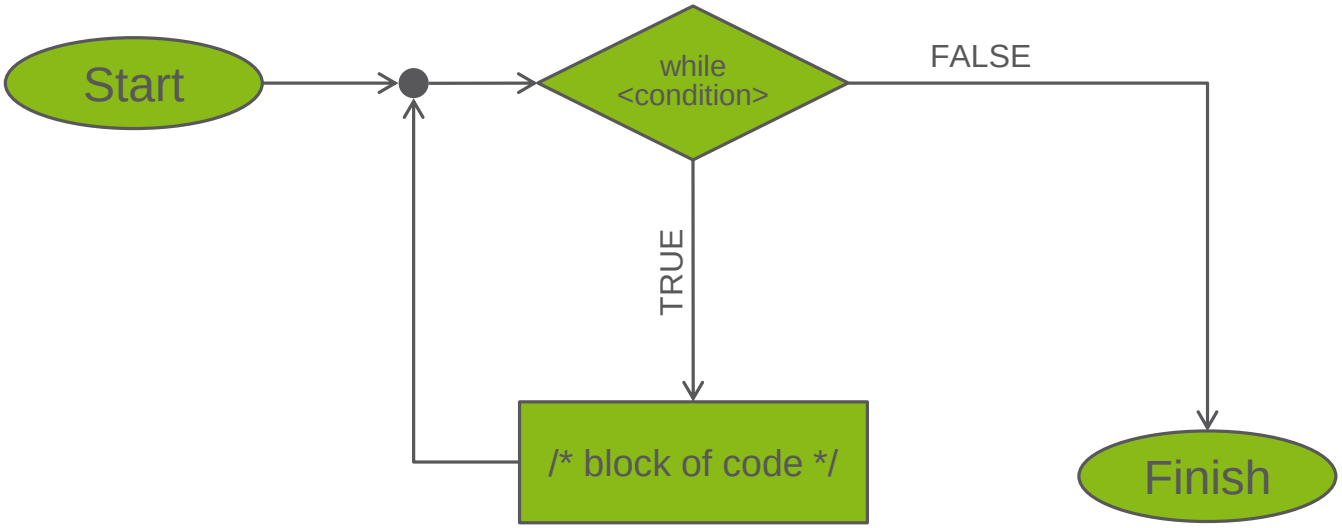
Switch-Case statement provides a mechanism similar to Else-If statement of comparing a single variable, placed after `switch`, with multiple values.

- Only one variable can be compared
- Compared variable type must be of integer type (`char, short, int, long, enum`)
- Once a comparison is true, further instructions are executed without comparing value
- In order to stop further comparing use `break` statement
- Default value is optional

```
switch  (expression)
{
  case constant1:
     group-of-statements-1;
     break; //break is optionally
  case constant2:
     group-of-statements-2;
     break;
  .
  .
  .
  default:
     default-group-of-statements
}
```

SWITCH

Loop **while** will execute as long as the condition is true.

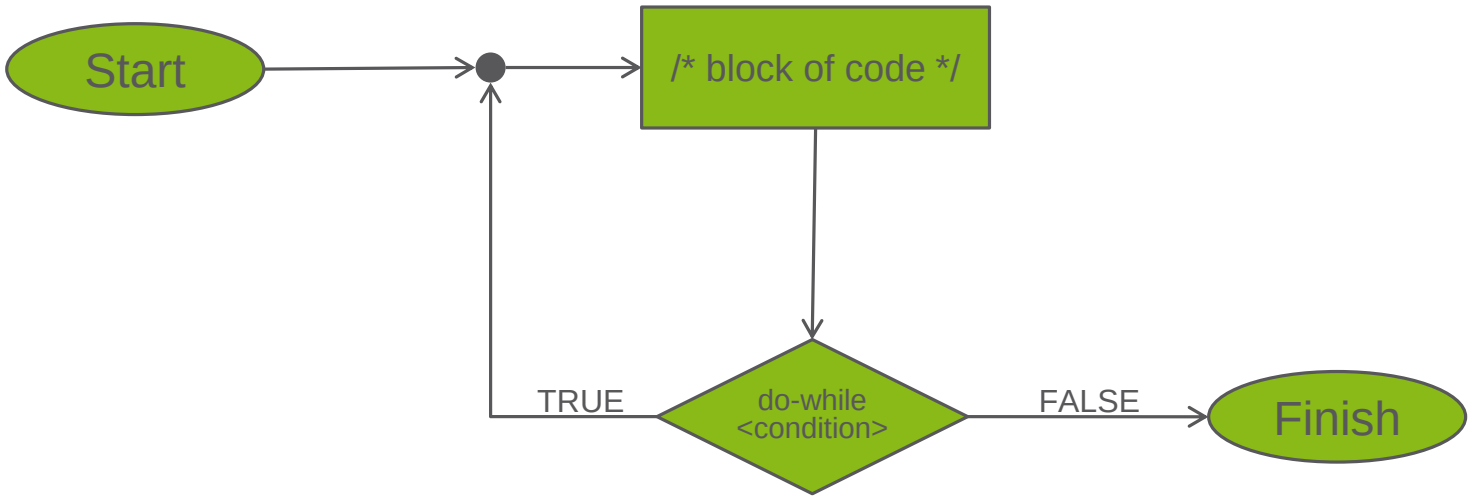



```
while(<condition>)
{
 /* block of code executed if condition is true */
}
```

Loop **while:**

- Condition is calculated **before** each loop
- Code block is executed as long as the condition is true

Loop **do–while** will execute the code block as long as the condition is true but at least once.

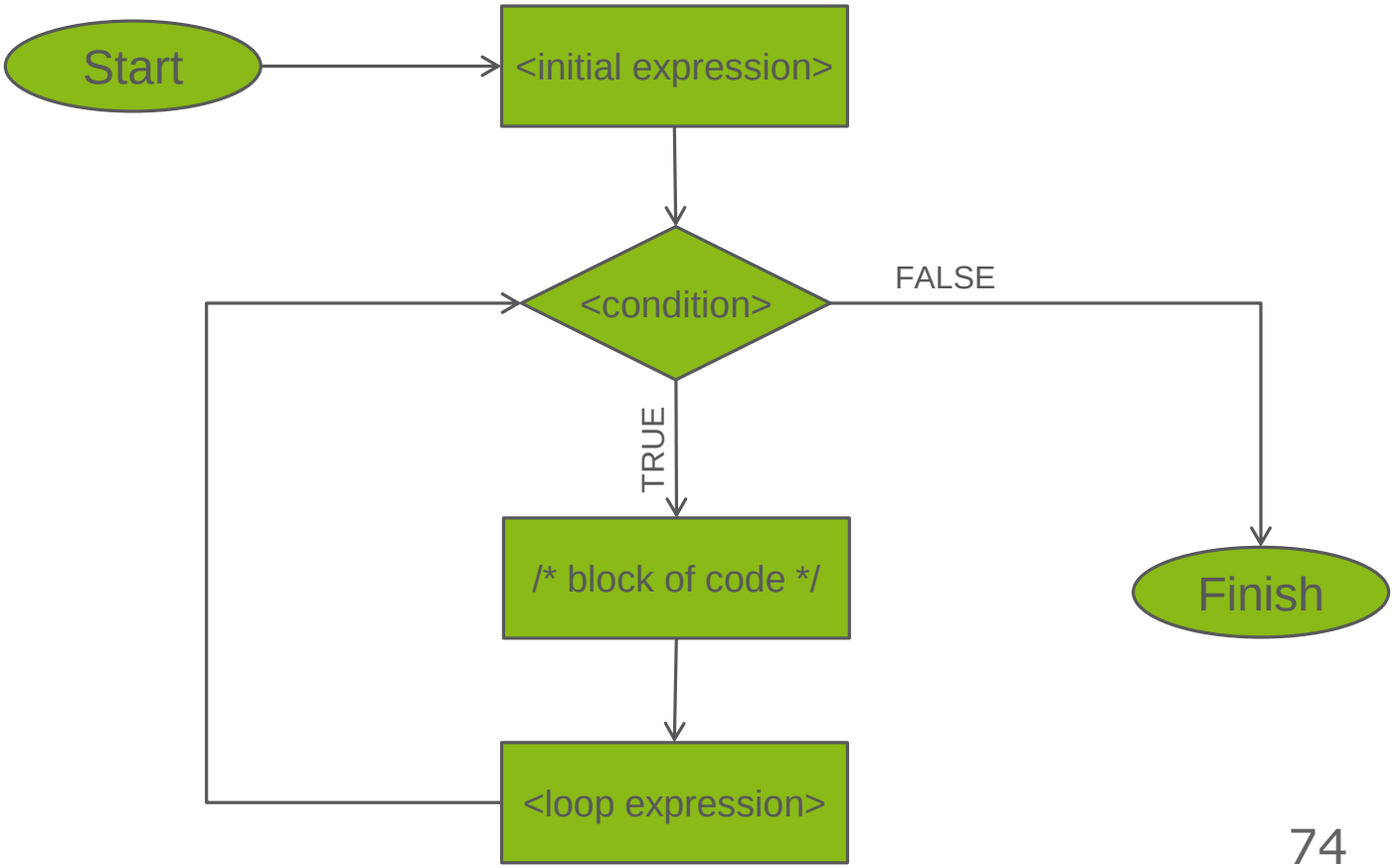


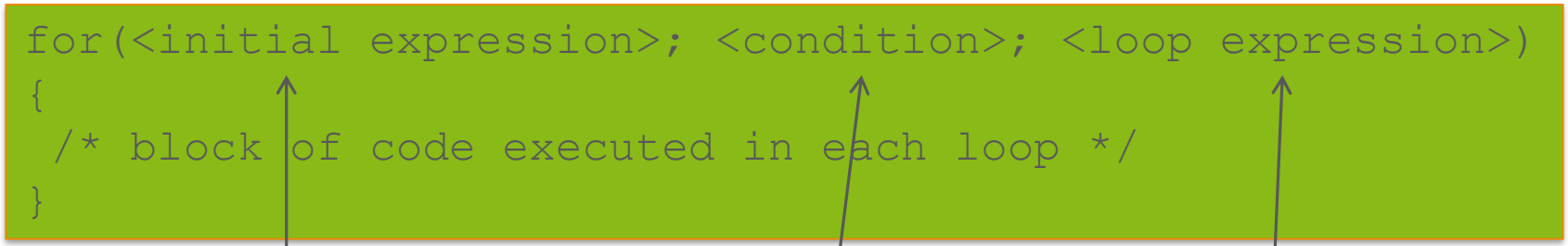

```
do
{
 /* block of code executed if condition is true */
} while(<condition>);
```

Loop **do–while**:

- Code block is executed for the first time **unconditionally**
- Condition is calculated before each loop from now on
- Code block is executed as long as the condition is true
- Requires semicolon (`;`) at the end of expression

Loop **for** has the build in <`initial expression`> executed once and an <`loop expression`> executed after each loop.





```
for(<initial expression>; <condition>; <loop expression>)
{
 /* block of code executed in each loop */
}
```

Initial Expression

Executed once **before** the looping begins.

Condition

Checked **before** each loop.

Loop Expression

Executed **after** each loop.

break statement terminates any loop. Program continues in the next statement after the loop.

continue statement is used to skip over the rest of the current loop iteration. The body of a loop is terminated immediately and next iteration of a loop begins.

```
for (int i = 0; i < 10; ++i)
{
  if (i < 4)
    continue;
  cout << i << endl;
  if (i > 6)
    break;
}
cout << "The end." << endl;
```

```
4
5
6
7
The end.
```

`goto` statement is used to immediately and unconditionally jump to another line of code. In order to use `goto` statement you must place a label at a point in your program.

```
bool isOK = true;
for (int i = 0; i < 10; ++i)
{
  /* some code here that may change isOK into 0*/
  if (isOK == false)
  {
    goto ERROR;
  }
}

ERROR:
cout << "There is an error!" << endl;
```

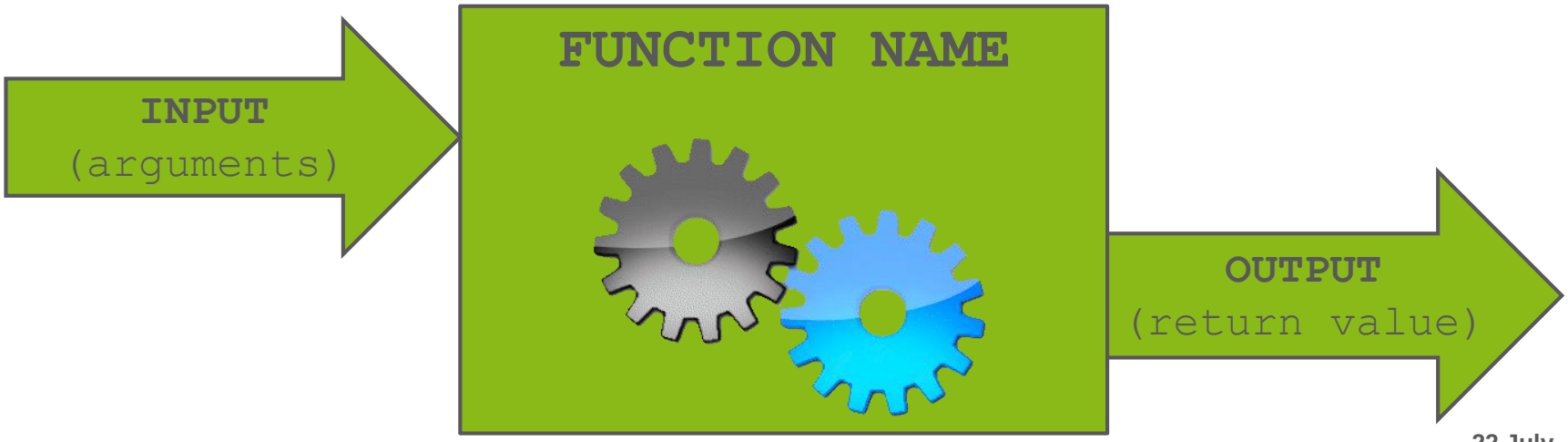
Summary:

- **Streams**
- **Branching**
  - If-Else
  - Switch-Case
  - Operator `?:`
- **Looping**
  - While
  - Do-While
  - For
  - `break`, `continue`, `goto`

CHAPTER IV
FUNCTIONS

Function is a block of code that:

- Has an identifier (name)
- Performs a specific task
- Can work with given parameters (arguments)
- Can return a single value



Function Declaration is used to inform the compiler of existence of a function and consist of:

- Function name
- Arguments types
- Return value type

```
<type returned> <name> (<argument1, argument2, …>);
```

Type Returned

If no data returned: `void`

Name

Also Address!

Argument

Just the types

Can be values

Can be references

Can have default values

Function Body is a block of code that implements the function.

```
<type returned> <name> (<parameter1, parameter2, …>)
{
    // function body
    return <return variable>;
}
```

Return Statement

Ends the function

Not required (`void`)

Multiple may occur

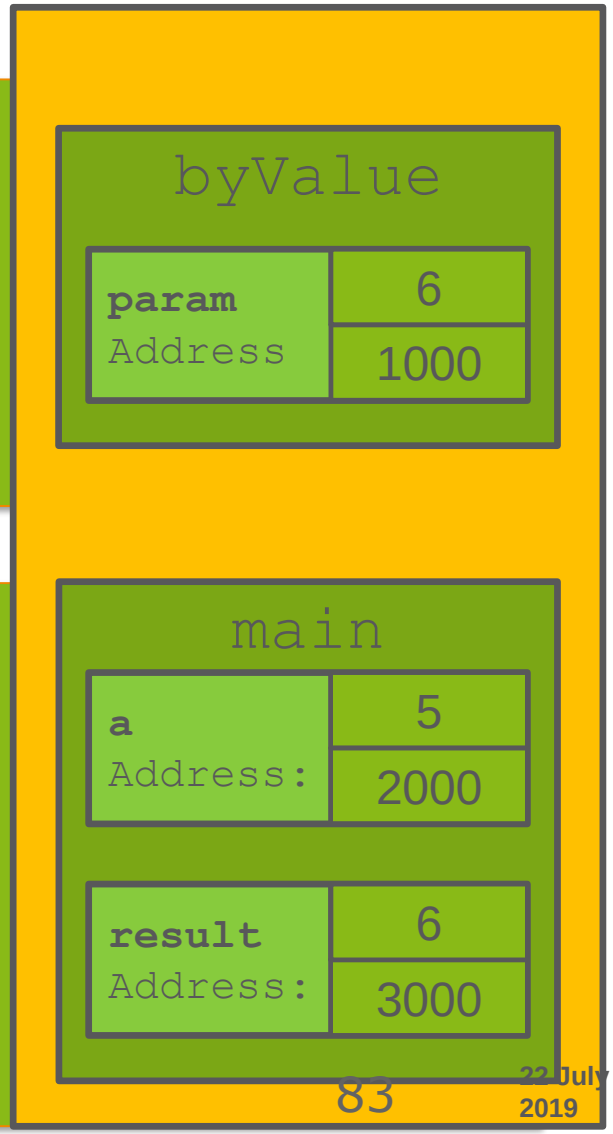
Parameter

Data Type + **Local** Name

Argument (by value)

```
int byValue(int param)
{
  param++;
  cout << param << endl;

  return param;
}
```

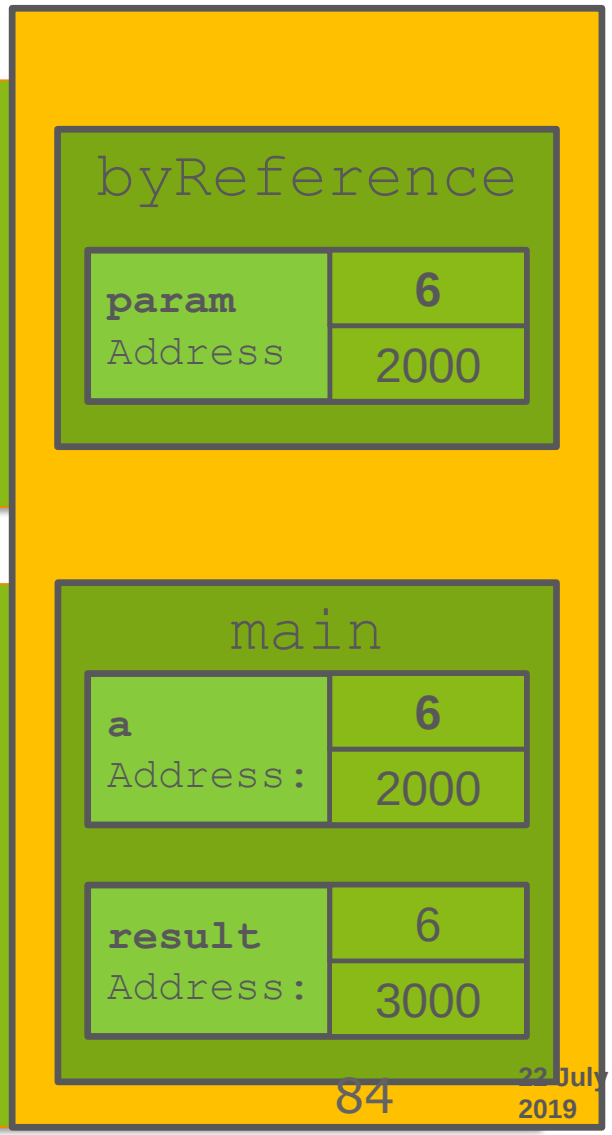
```
int main()
{
  int a = 5, result;
  cout << a << endl;
  result = byValue(a);
  cout << a << endl;

  return 0;
}
```



Argument (by reference)

```
int byReference(int &param)
{
  param++;
  cout << param << endl;

  return param;
}
```

```
int main()
{
  int a = 5, result;
  cout << a << endl;
  result = byReference(a);
  cout << a << endl;

  return 0;
}
```



Function can return no value and then:

- It's return type should be `void`
- No `return` statement is required

```
void printFunction(int a, int b)
{
    cout << a << endl;
    cout << b << endl;
}
```

Default Values in function parameters:

- Default value will be used when the corresponding argument is left blank when calling the function
- Only the rightmost arguments can be set as default

```
#include <iostream>

using namespace std;

double divide (double a, double b = 2)
{
  return a/b;
}

int main ()
{
  cout << divide(12) << endl;
  cout << endl;
  cout << divide(20,4) << endls;
  return 0;
}
```

Default Value
b = 2

Same as:
cout << divide(12, 2);

Function name overloading:

- Functions can have the same name if their parameter types or their number is different

Inline functions are declared using `inline` keyword:

- Does not change the behaviour of a function
- The compiler inserts the body of the function instead of calling it

```
inline <type returned> <name> (<param1, param2, …>)
{
    // function body
    return <return variable>;
}
```

`main()` is a special function where a program starts execution.

Each program must have exactly one `main()` function.

```
int main (int argc, char *argv[])
{
  // Body of main()
}
```

`int argc` – the number of arguments entered by the user during program start.

`char * argv[]` – arguments entered by the user during program start

Rules of declaring and reading complex variable declarations are simple.

- Start with the variable name
- Bounce from the variable name right to left until all elements are used

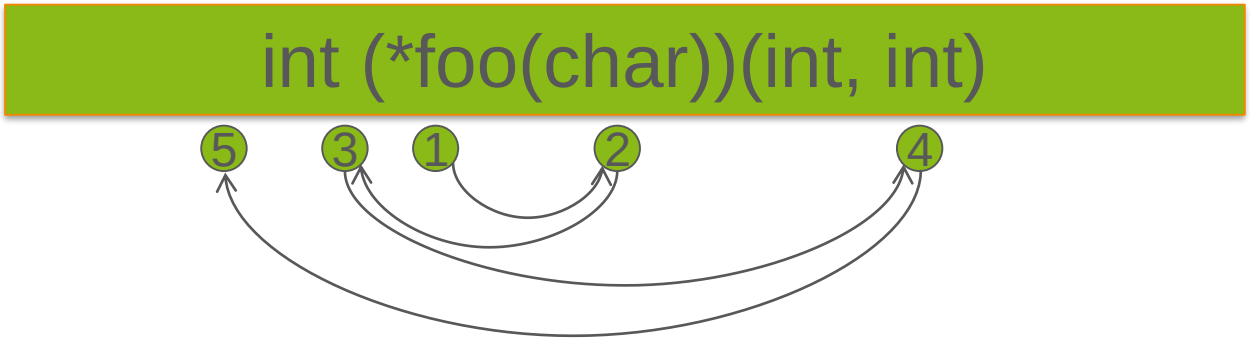
foo is ...

foo is array of 5...

foo is array of 5 pointers to...

foo is array of 5 pointers to pointer to...

foo is array of 5 pointers to pointer to long



foo is ...

... function with one parameter (char) ...

... that returns pointer to...

... function with two parameters (int, int) ...

... that returns int

Summary:

- **Functions**
  - Definition: return type, name, parameters
  - Sending by value
  - Sending by reference
  - Default values for parameters
  - Function name overloading
  - `inline`
  - `void` / `return` statement

CHAPTER V:
COMPOUND DATA TYPES

Array is a box which holds multiple variables of the same type.

- Helps organize the variables
- Variables stored in an array are called elements
- Elements are indexed from 0
- All elements share the same name
- Name of an array is the address of its first element

```
<data type> <name> [<size>];
```

Data Type

All elements will have the same type.

Size

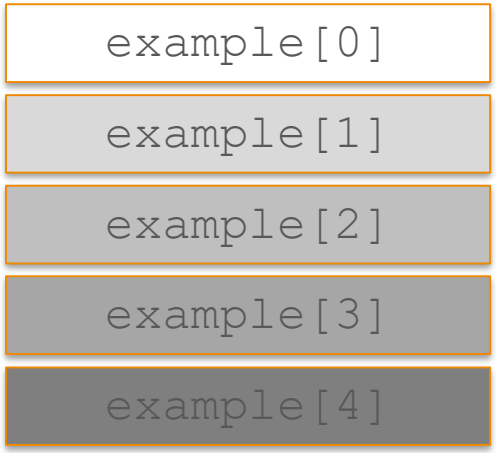
Number of elements in array (must be known and can't change).

COMPOUND DATA TYPES

```
short example[5];
```

STACK

1500	1501	1502	1503
1504	1505	1506	1507
1508	1509	1510	1511
1512	1513	1514	1515
1516	1517	1518	1519
example[0]		example[1]	
example[2]		example[3]	
example[4]		1530	1531
1532	1533	1534	1535



example array

5 elements.

Solid block of memory.

example = 1520 (address!)

&example = 1520 (address!)

It is possible to initialize an array during declaration.

```
int array[5] = {0, 2, 4, 6, 8};
```

The array size may be omitted but the array must be initialized during such declaration.

```
int array[] = {0, 2, 4, 6, 8};
```

Declarations above will result in the same array.

- Global arrays elements are initialized with their default values (zeros for fundamental types)
- Arrays of local scope are not initialized by any default value

Array elements can be accessed using array index.

The `table[i]` refers to the i^{th} element of an array `table`.

```
int values[5] = {0, 2, 4, 6, 8};
for (int i = 0; i < 5; ++i)
{
  cout << values[i] << endl;
}
```

```
int values[5];
for (int i = 0; i < 5; ++i)
{
  cin >> values[i];
}
```

Multi-Dimensional arrays are in fact arrays of arrays.

array[2][5]

int	int	int	int	int
int	int	int	int	int

```
int array[2][5] = { {0, 2, 4, 6, 8}, {1, 2, 3, 4, 5} };

int x, y;
for (x = 0; x < 2; ++x)
{
  for (y = 0; y < 5; ++y)
  {
    cout << "array[" << x << "][" << y << "]=" << array[x][y];
  }
}
```

Structure is a collection of one or more variables:

- Variables can be of different types
- Grouped together under a single name for convenient handling
- Size of structure is no less than sum of sizes of variables it holds

Union is a storage of variables:

- Variables can be of different types
- Grouped together under a single name for convenient handling
- Only one variable can be stored at a time
- Size of union equals the size of the largest variable the union can storage


```
string title
```

```
string author
```

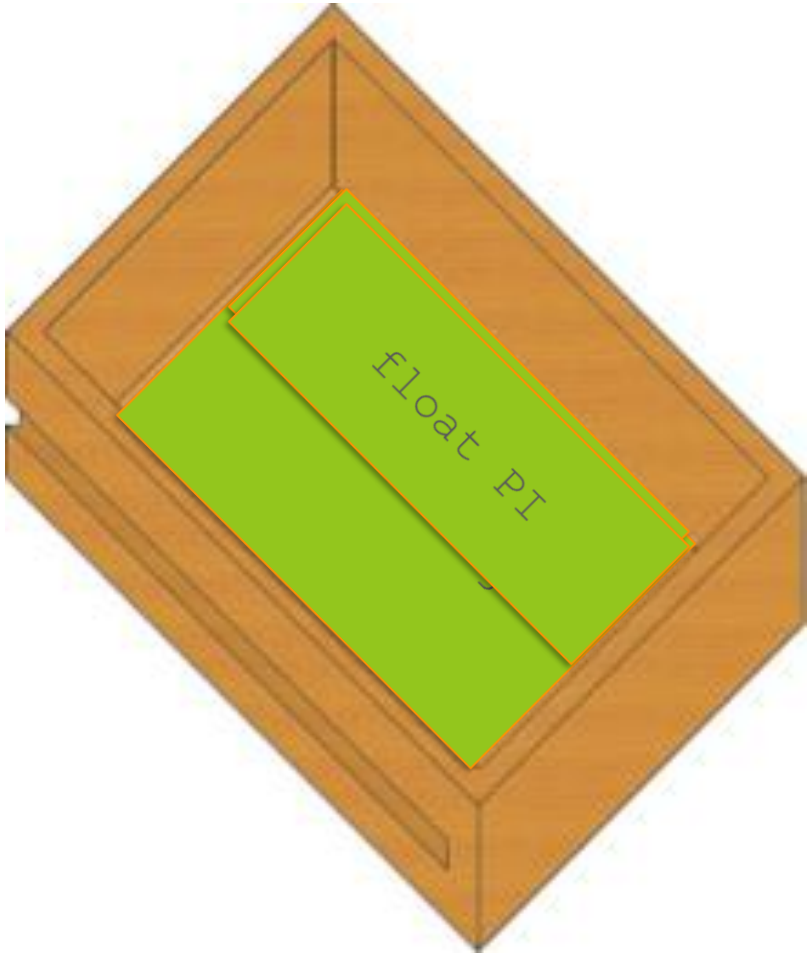
```
int pages
```

```
float price
```

Structure

All data available.

COMPOUND DATA TYPES



double large

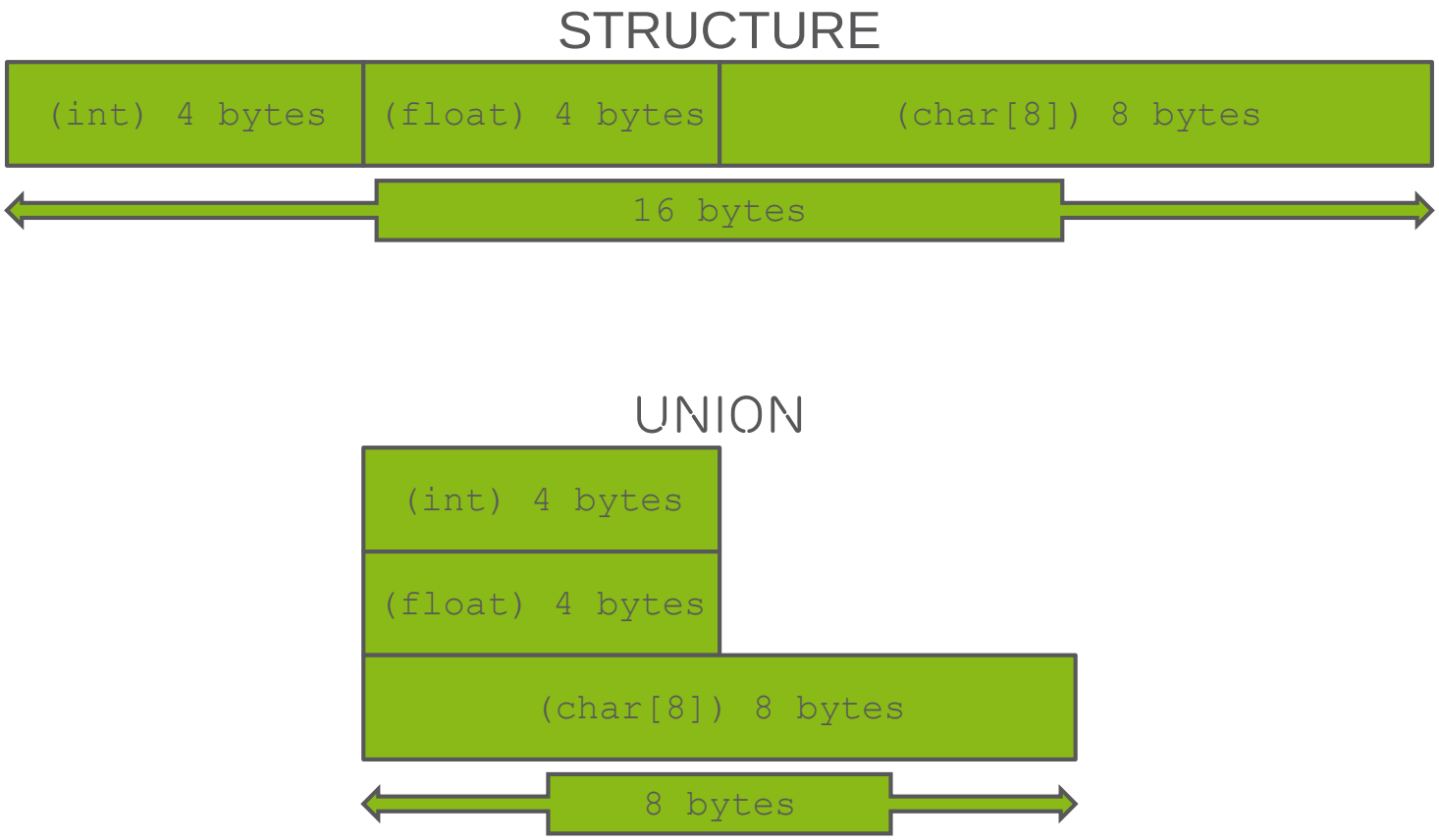
int number

float PI

Union

Only one data available.

Structures and Unions memory usage.



Structures are declared using a keyword `struct`.

```
struct <structure name>
{
  <structure fields>
};
```

Structures fields are variables that structure holds.

```
struct book
{
  string title;
  string author;
  int pages;
  float price;
};

book myBook;
```

=

```
struct book
{
  string title;
  string author;
  int pages;
  float price;
} myBook;
```

Structure initialization looks very similar to initialization of an array.

```
struct book
{
  string title;
  string author;
  int pages;
  float price;
};

book myBook = {
  “Lord of The Rings – Return of The King”,
  “JRR Tolkien”,
  565,
  19.5
};
```

Structure elements can be accessed by using (.) (dot) operator.

```
cout << "Title: " << myBook.title << endl;
cout << "Author: " << myBook.author << endl;
cout << "Pages: " << myBook.pages << endl;
cout << "Price" << myBook.price << endl;
```

```
getline(cin, myBook.title);
getline(cin, myBook.author);
cin >> myBook.pages;
cin >> myBook.price;
```

Structures can contain other structures as members.

```
struct Person
{
  int age;
  struct Date
  {
    int year;
    int month;
    int day;
  } birth;

};

Person thomas;
thomas.age = 28;
thomas.birth.year = 1984;
thomas.birth.month = 10;
thomas.birth.day = 8;
```

Pointer to a structure may be defined the same way as for any basic variable type. The only difference is in the way the structure elements are accessed.

Structure elements can be accessed from a pointer by using the (->) (arrow) operator.

```
book myBook = { "Title", "Author", 100, 10.5 };
book *ptrBook = &myBook;

cout << "Title: " << ptrBook->title << endl;
cout << "Author: " << ptrBook->author << endl;
cout << "Pages: " << ptrBook->pages << endl;
cout << "Price" << ptrBook->price << endl;
```

Unions are declared using a keyword `union` the same way as structures are.

```
union <union name>
{
  <union fields>
};
```

Union fields are variables that union may hold.

```
union multiBox
{
  int iValue;
  float fValue;
  char cValue;
};

union mutiBox box;
```

=

```
union multiBox
{
  int iValue;
  float fValue;
  char cValue;
} box;
```

Union can store only one of its elements at a time and only the first element can be initialized.

```
union multiBox
{
  int iValue;
  float fValue;
  char cValue;
};

multiBox box1 = {4};        // iValue = 4
multiBox box2 = {4.3};      // iValue = 4
multiBox box3 = {'c'};      // iValue = 99 ('c')
```

Summary:

- **Array**
- **Union**
  - All variables in the same memory area
- **Structure**
  - Variables in separate memory area

CHAPTER VI
DYNAMIC MEMORY ALLOCATION

Dynamic Memory Allocation means an allocation of memory as required at runtime:

- Dynamically allocated memory exists until it is released explicitly by the programmer
- Allows the programmer to handle variables and data structures which are by nature dynamic
- Allows for efficient use of the memory
- Requires extra caution
- Memory is allocated on heap

Requesting dynamic memory is handled via operator `new`, followed by a data type identifier.

```
<data type> * <pointer name> = new <data type>;
```

```
int *i = new int;
```

Sequence of one or more elements of the same type is required by following operator `new` by brackets (`[]`) with the number or required elements.

```
<pointer> = new <data type> [<number of elements>];
```

```
int *i = new int[5];
```

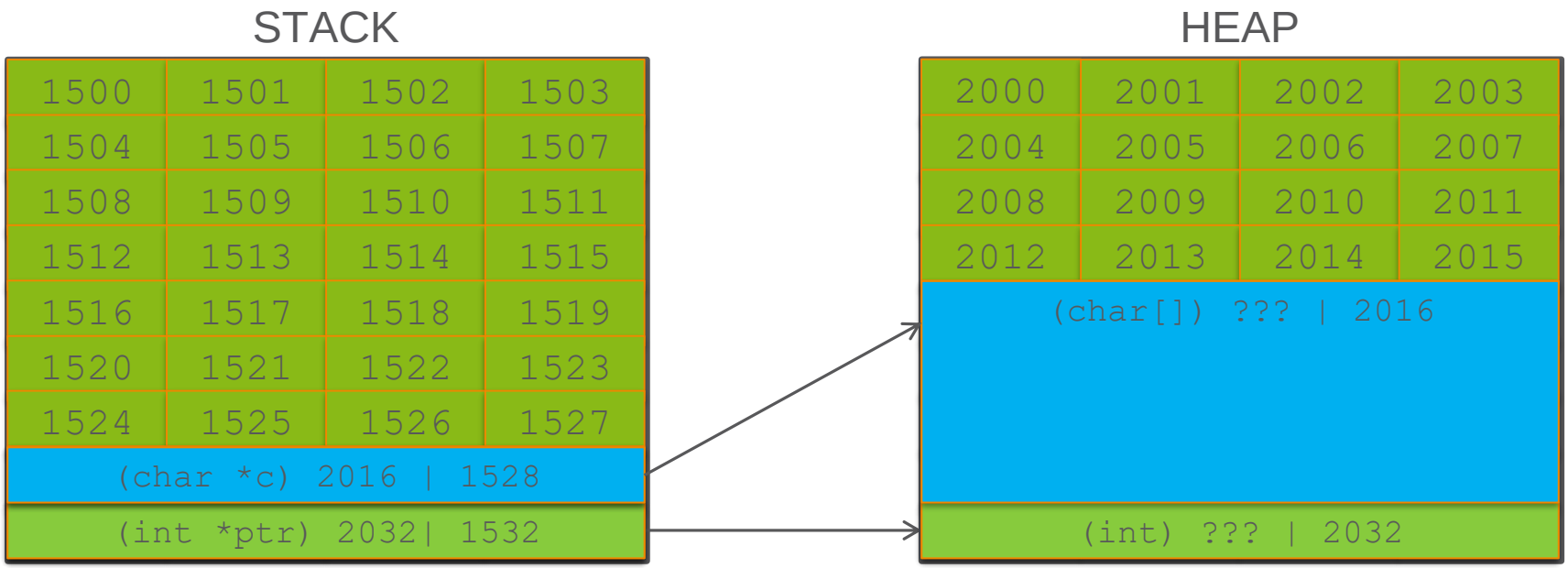
Memory allocated via operator `new` reside in heap.

```
int *ptr;
```

```
ptr = new int;
```

```
char *c;
```

```
c = new char[16];
```



Operator `new` allocates memory. If the allocation fails `bad_alloc` exception is thrown.

- Exception must be handled – otherwise application will be terminated

In order not to throw an exception in case of allocation failure `(nothrow)` method can be used.

```
int *i = new (nothrow) int[5];
```

In this case, if the allocation of this block of memory failed, the failure could be detected by checking if i took a NULL pointer value:

Allocated memory, once no longer needed, should be freed using operator `delete`, so that the memory becomes available again.

`int *i = new int;`	`allocation - new`
`...`	`operations on pointer`
`delete i;`	`deallocation - delete`

Sequence of one or more elements allocated using operator `new[]` must be deallocated using `delete[]` operator.

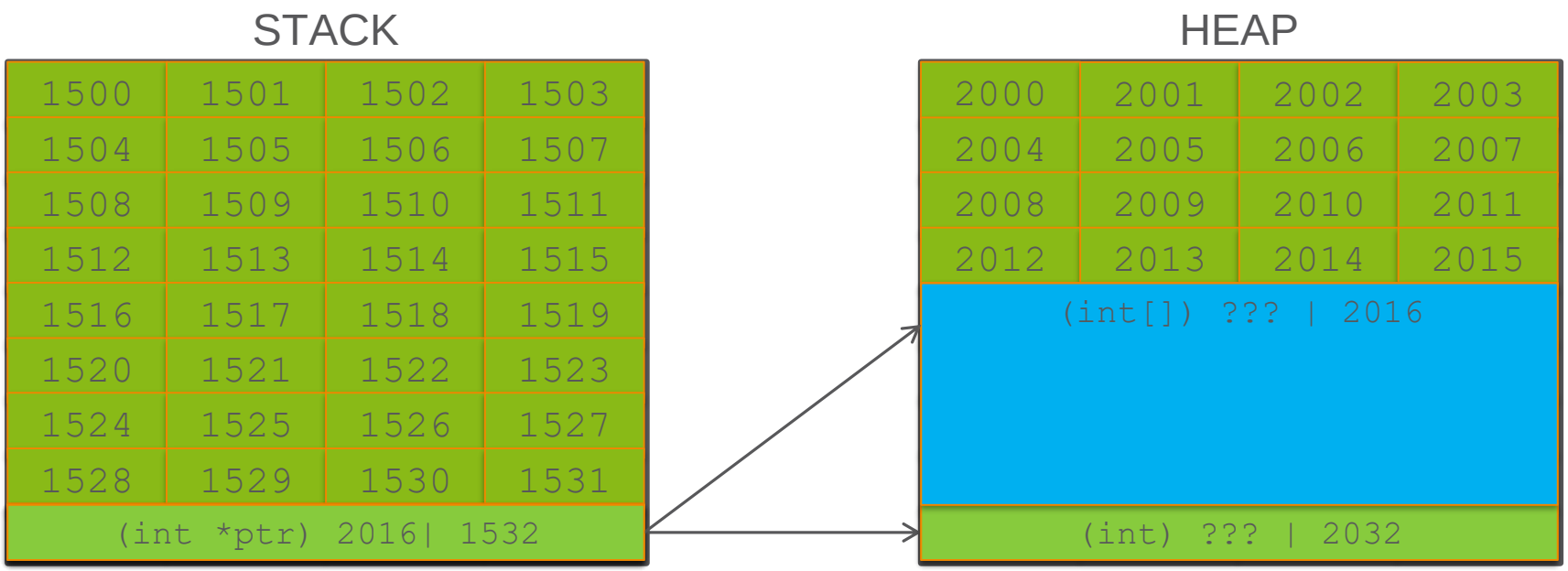
`int *i = new int[5];`	`allocation - new[]`
`...`	`operations on pointer`
`delete[] i;`	`Deallocation - delete[]`

DYNAMIC MEMORY ALLOCATION

```
int *ptr;
```

```
ptr = new int;
```

```
ptr = new int[4];
```



Memory Lost – memory leak!

Summary:

- **Allocation**
  - Operators `new` and `new[]`
- **Freeing Memory**
  - Operators `delete` and `delete[]`
- **Troubles**
  - Memory Leak
  - Exceptions in Memory Allocation

Thank you for your attention