

These are the slides for the Beginning C++ Programming - From Beginner to Beyond on Udemy. They are provided free of charge to all students.

More information about the course: <https://lpa.dev/u1bcppp>

If you have any questions or queries, please add your feedback in the Q&A section of the course on Udemy.

Best regards,

Tim Buchalka
Learn Programming Academy

Beginning C++ Programming Slides

Main Course Slides.

Welcome and Introduction to the Course

- About me
- My assumptions
- Your background
- The curriculum and Modern C++
- Practice!
- Please ask questions

Why Learn C++?

- Popular
 - Lots of code is still written in C++
 - Programming language popularity indexes
 - Active community, GitHub, stack overflow
- Relevant
 - Windows, Linux, Mac OSX, Photoshop, Illustrator, MySQL, MongoDB, Game engines, more ...
 - Amazon, Apple, Microsoft, PayPal, Google, Facebook, MySQL, Oracle, HP, IBM, more...
 - VR, Unreal Engine, Machine learning, Networking & Telecom, more...
- Powerful
 - fast, flexible, scalable, portable
 - Procedural and Object-oriented
- Good career opportunities
 - C++ skills always in demand
 - C++ = Salary++

Modern C++ and the C++ Standard

- Early 1970s
 - C Programming Language
 - Dennis Ritchie
- 1979
 - Bjarne Stroustrup
 - C with Classes
- 1983
 - Name changed to C++
- 1989
 - First commercial release
- 1998
 - C++98 Standard
- 2003
 - C++03 Standard
- 2011
 - **C++11 Standard**
- 2014
 - **C++14 Standard**
- 2017
 - **C++17 Standard**

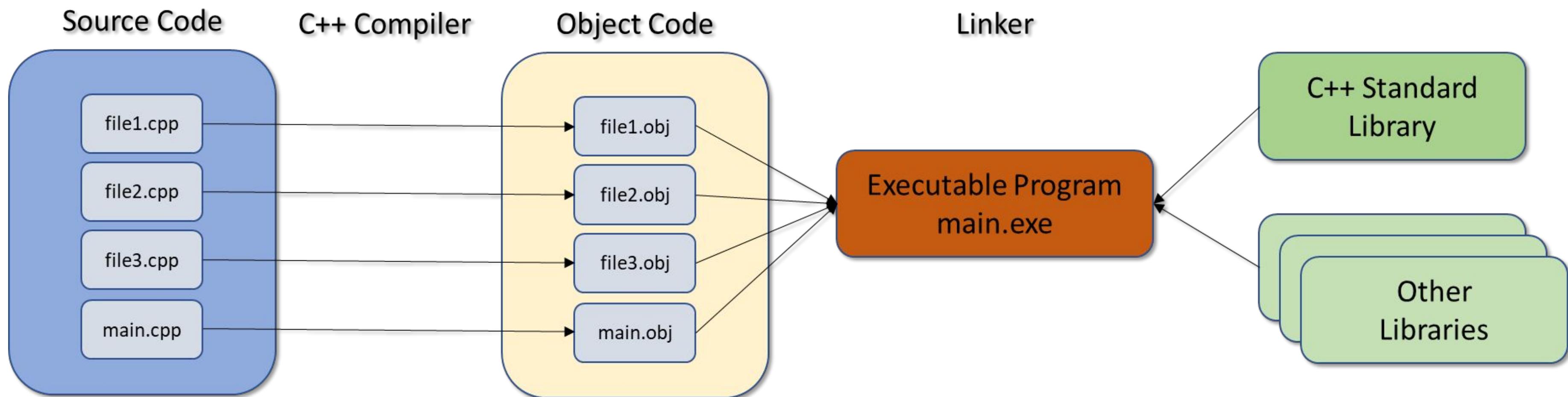
Modern C++ and the C++ Standard

- Classical C++
 - Pre C++11 Standard
- Modern C++
 - C++11
 - Lots of new features
 - C++14
 - Smaller changes
 - C++17
 - Simplification
 - Best practices
 - Core Guidelines

How does it all work?

- You must tell the computer EXACTLY what to do
 - Program – like a recipe
- Programming language
 - source code
 - high-level
 - for humans
- Editor – used to enter program text
 - .cpp and .h files
- Binary or other low-level representation
 - object code
 - for computers
- Compiler – translates from high-level to low-level
- Linker – links together our code with other libraries
 - Creates executable program
- Testing and Debugging – finding and fixing program errors

The C++ Build Process



Integrated Development Environments (IDEs)

- Editor
- Compiler
- Linker
- Debugger
- Keep everything in sync
- CodeLite
- Code::Blocks
- NetBeans
- Eclipse
- CLion
- Dev-C++
- KDevelop
- Visual Studio
- Xcode

Section Overview

- Microsoft Windows, Mac OSX, Ubuntu Linux 18.04
 - C++ Compiler
 - CodeLite Integrated Development Environment (IDE)
 - Configure CodeLite
 - Create a Default CodeLite Project Template
- Using the Command-line
- Using a Web-based compiler



Please check the **Resources** for the videos as I will post updates there as needed.

Using the command-line interface

- A text editor (not a Word Processor)
 - A command-prompt or terminal window
 - An installed C++ compiler
-
- No need for an IDE
 - Simple, efficient workflow
 - Better as you gain experience
 - Can be used if you are overwhelmed by IDEs
 - Useful if hardware resources are limited

Curriculum Overview

- Getting Started
- Structure of a C++ Program
- Variables and Constants
- Arrays and Vectors
- Strings in C++
- Expressions, Statements and Operators
- Statements and Operators
- Determining Control Flow
- Functions
- Pointers and References
- OOP – Classes and Objects
- Operator Overloading
- Inheritance
- Polymorphism
- Smart Pointers
- The Standard Template Library (STL)
- I/O Streams
- Exception Handling

Curriculum Overview

Challenge Exercises

- At the end of most course sections
- Develop real C++ programs using what we discussed in the section
- Section challenges
 - Description
 - Starting project
 - Completed solution
- Have fun and keep coding!

Curriculum Overview

Quizzes

- At the end of most sections
- Reinforce the concepts learned in each section
- Quiz style
 - Multiple choice
 - Fill in blank
 - Concept oriented vs. code oriented

Section Overview

- CodeLite IDE Quick Tour
- Our first program
 - Building
 - Running
 - Errors
 - Warnings

Compiler Errors

- Programming languages have rules
- Syntax errors – something wrong with the structure

```
std::cout << "Errors" << std::endl;
```

```
return 0
```

- Semantic errors – something wrong with the meaning
- a + b; When it doesn't make sense to add a and b

Compiler Warnings

Do NOT ignore them!

- The compiler has recognized an issue with your code that could lead to a potential problem
- It's only a warning because the compiler is still able to generate correct machine code

```
int miles_driven;  
std::cout << miles_driven;
```

warning: 'miles_driven' is used uninitialized ...

Linker Errors

- The linker is having trouble linking all the object files together to create an executable
- Usually there is a library or object file that is missing

Runtime Errors

- Errors that occur when the program is executing
- Some typical runtime errors
 - Divide by zero
 - File not found
 - Out of memory

Can cause your program to 'crash'

Exception Handling can help deal with runtime errors

Logic Errors

- Errors or bugs in your code that cause your program to run incorrectly
- Logic errors are mistakes made by the programmer

Suppose we have a program that determines if a person can vote in an election and you must be 18 years or older to vote.

```
if (age > 18) {  
    std::cout << "Yes, you can vote!";  
}
```

Test your code!!

Section Overview

The Structure of a C++ Program

- Basic Components
- Preprocessor Directives
- The main function
- Namespaces
- Comments
- Basic I/O

The Structure of a C++ Program

Components of a C++ Program

- Keywords
- Identifiers
- Operators
- Punctuation
- Syntax

Keywords

- Have special meaning in C++
- Are reserved by the C++ language

```
#include <iostream>

int main() {

    int favorite_number;

    std::cout << "Enter your favorite number between 1 and 100: ";
    std::cin >> favorite_number;

    std::cout << "Amazing!! That's my favorite number too!" << std::endl;

    std::cout << "No really!!, " << favorite_number << " is my favorite number!" << std::endl;

    return 0;
}
```

Identifiers

- Programmer-defined names
- Not part of the C++ language
- Used to name variables, functions, etc.

```
#include <iostream>

int main() {
    int favorite_number;
    std::cout << "Enter your favorite number between 1 and 100: ";
    std::cin >> favorite_number;
    std::cout << "Amazing!! That's my favorite number too!" << std::endl;
    std::cout << "No really!!, " << favorite_number << " is my favorite number!" << std::endl;
    return 0;
}
```

Operators

- Arithmetic operators, assignment, <<, >>
- Are reserved by the C++ language

```
#include <iostream>

int main() {

    int favorite_number;

    std::cout << "Enter your favorite number between 1 and 100: ";
    std::cin >> favorite_number;

    std::cout << "Amazing!! That's my favorite number too!" << std::endl;
    std::cout << "No really!!, " << favorite_number << " is my favorite number!" << std::endl;

    return 0;
}
```

Punctuation

- Special characters that separate, terminate items

```
#include <iostream>

int main() {

    int favorite_number;

    std::cout << "Enter your favorite number between 1 and 100: ";
    std::cin >> favorite_number;

    std::cout << "Amazing!! That's my favorite number too!" << std::endl;
    std::cout << "No really!!, " << favorite_number << " is my favorite number!" << std::endl;

    return 0;
}
```

Syntax

- How the programming elements are put together to form a program
- Programming languages have rules

```
#include <iostream>

int main() {

    int favorite_number;

    std::cout << "Enter your favorite number between 1 and 100: ";
    std::cin >> favorite_number;

    std::cout << "Amazing!! That's my favorite number too!" << std::endl;

    std::cout << "No really!!, " << favorite_number << " is my favorite number!" << std::endl;

    return 0;
}
```

Our Section 4 Challenge Solution

Modified slightly

- Added comments
- Using namespace instead of std::

```
// Preprocessor directive that include the iostream library headers
#include <iostream>

// use the std namespace
using namespace std;

/* Start of the program
 * program execution always begins with main()
 */
int main() {

    int favorite_number;      // declare my favorite number variable

    // Note that I'm no longer using std::
    // Prompt the user to enter their favorite number

    cout << "Enter your favorite number between 1 and 100: ";

    // read the user's input into the variable favorite_number
```

Our Section 4 Challenge Solution

Modified slightly

```
// Preprocessor directive that include the iostream library headers
#include <iostream>

// use the std namespace
using namespace std;

/* Start of the program
 * program execution always begins with main()
 */
int main() {

    int favorite_number;    // declare my favorite number variable

    // Note that I'm no longer using std::
    // Prompt the user to enter their favorite number

    cout << "Enter your favorite number between 1 and 100: ";

    // read the user's input into the variable favorite_number
    cin >> favorite_number;

    // Out put the results to the user
    cout << "Amazing!! That's my favorite number too!" << endl;
    cout << "No really!!, " << favorite_number << " is my favorite number!" << endl;

    return 0;
}
```

Preprocessor Directives

- What is a preprocessor?
- What does it do?
- Directives start with '#'
- Commands to the preprocessor

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

```
#if
#elif
#else
#endif
```

Comments

- Ignored by the compiler
- Used to explain your code
- Two styles

```
int favorite_number; // This will store my favorite number

// The following lines convert Euros to US Dollars

/* This is a comment
   that spans multiple lines.
   All of these lines will be ignored by the compiler
*/

// Why should we have to explain our own code?
```

Functions

- main() is a required function in C++
- Break up your code into units of functionality
- Can optionally receive and return information

```
/* This is a function that expects two integers a and b
   It calculates the sum of a and b and returns it to the caller
   Note that we specify that the function returns an int
*/
int add_numbers(int a, int b)
{
    return a + b;
}

// I can call the function and use the value that is returns

cout << add_numbers(20, 40);
```

Preprocessor Directives

- What is a preprocessor?
- What does it do?
- Directives start with '#'
- Commands to the preprocessor

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

#if
#elif
#else
#endif

#ifndef
#ifndef
#define
#undef

#line
#error
#pragma
```

The main() function

- Every C++ program must have exactly 1 main() function
- Starting point of program execution
- return 0 indicates successful program execution
- 2 versions that are both valid

```
int main()
{
    // code

    return 0;
}

program.exe

int main(int argc, char *argv[])
{
    // code

    return 0;
}

program.exe argument1 argument2
```

Namespaces

- Why `std::cout` and not just `cout`?
- What is a naming conflict?
- Names given to parts of code to help reduce naming conflicts
- `std` is the name for the C++ ‘standard’ namespace
- Third-party frameworks will have their own namespaces
- Scope resolution operator `::`
- How can we use these namespaces?

Explicitly using namespaces

```
#include <iostream>

int main()
{
    int favorite_number;

    std::cout << "Enter your favorite number between 1 and 100: ";

    std::cin >> favorite_number;

    std::cout << "Amazing!! That's my favorite number too!" << std::endl;
    std::cout << "No really!!, " << favorite_number
        << " is my favorite number!" << std::endl;

    return 0;
}
```

The using namespace directive

```
#include <iostream>

using namespace std;      // Use the entire std namespace

int main()
{
    int favorite_number;

    cout << "Enter your favorite number between 1 and 100: ";

    cin >> favorite_number;

    cout << "Amazing!! That's my favorite number too!" << endl;
    cout << "No really!!, " << favorite_number
        << " is my favorite number!" << endl;

    return 0;
}
```

Qualified using namespace variant

```
#include <iostream>

using std::cout; // use only what you need
using std::cin;
using std::endl;

int main()
{
    int favorite_number;

    cout << "Enter your favorite number between 1 and 100: ";

    cin >> favorite_number;

    cout << "Amazing!! That's my favorite number too!" << endl;
    cout << "No really!!, " << favorite_number
        << " is my favorite number!" << endl;

    return 0;
}
```

Basic I/O using cin and cout

cout, cin, cerr, and clog are objects representing streams

cout

- standard output stream
- console

cin

- standard input stream
- keyboard

<<

- Insertion operator
- output streams

>>

- extraction operator
- input streams

cout and <<

- Insert the data into the cout stream

```
cout << data;
```

- Can be chained

```
cout << "data 1 is " << data1;
```

- Does not automatically add line breaks

```
cout << "data 1 is " << data1 << endl;
```

```
cout << "data 1 is " << data1 << "\n";
```

cin and >>

- Extract data from the cin stream based on data's type

```
cin >> data;
```

- Can be chained

```
cin >> data1 >> data2;
```

- Can fail if the entered data cannot be interpreted

data could have an undetermined value

Section Overview

Variables and Constants

- Declaring variables
- C++ primitive types
 - integer
 - floating point
 - boolean
 - character
- sizeof operator

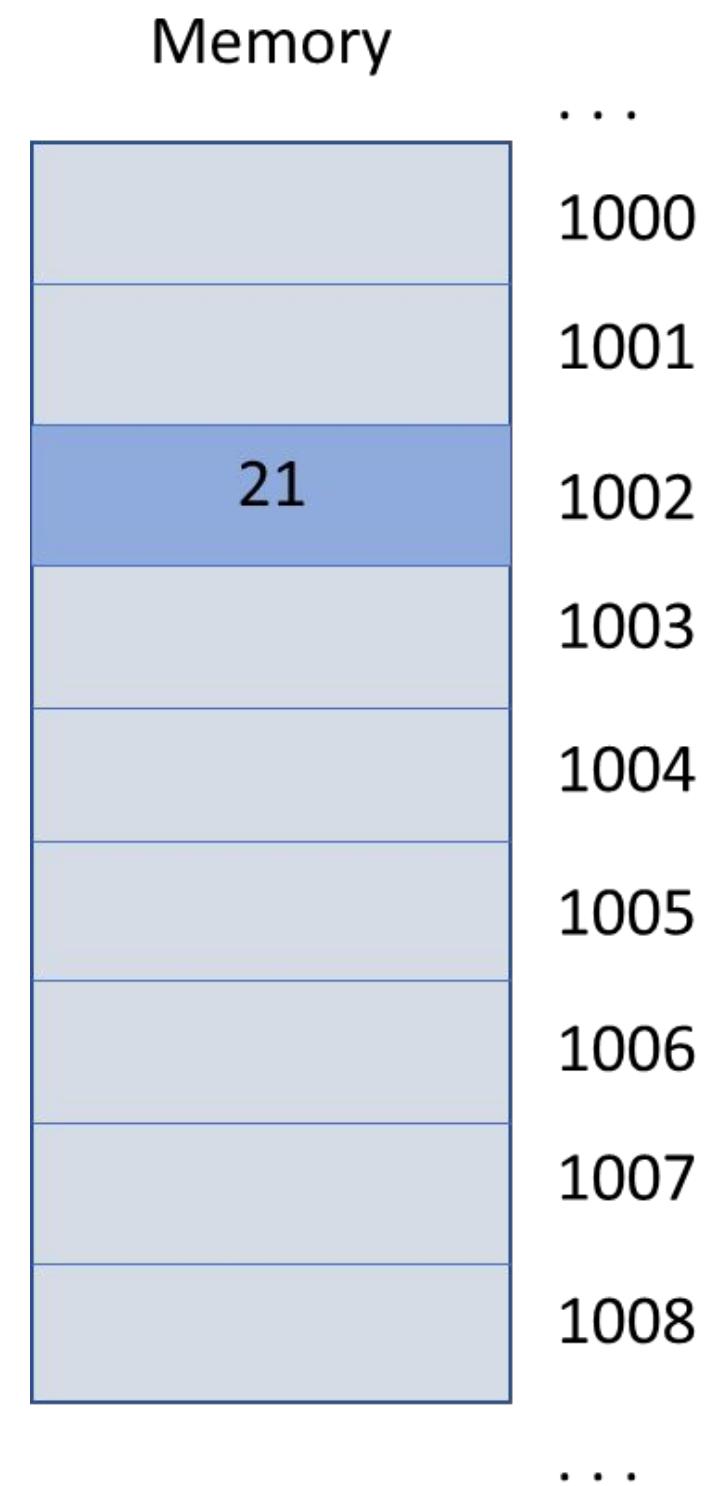
Section Overview

Variables and Constants

- What is a constant?
- Declaring constants
- Literal constants
- Constant expressions

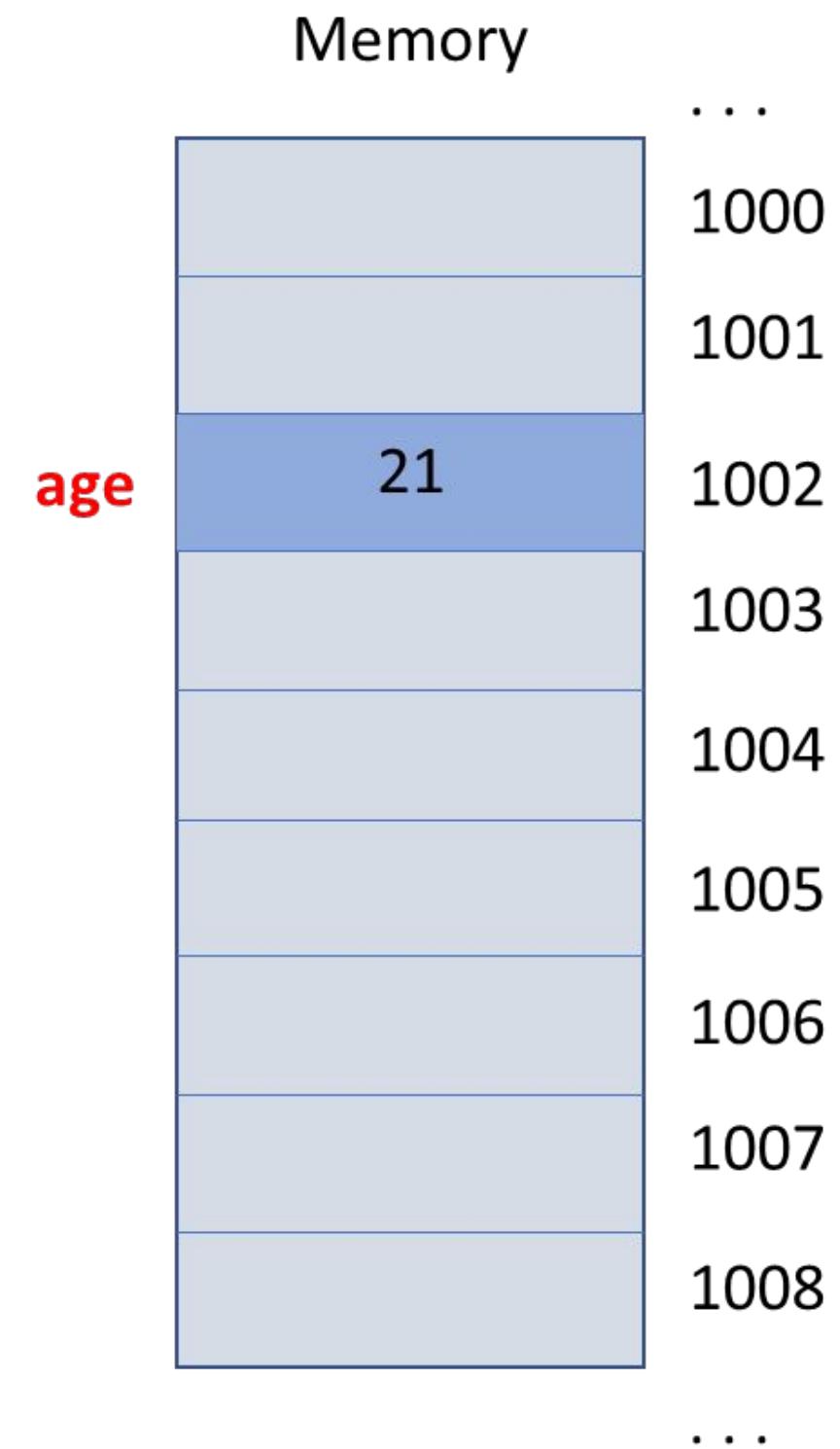
What is a variable?

move 21 to location 1002



What is a variable?

move 21 to age



What is a variable?

- A variable is an abstraction for a memory location
- Allow programmers to use meaningful names and not memory addresses
- Variables have
 - Type – their category (integer, real number, string, Person, Account...)
 - Value – the contents (10, 3.14, "Frank"...)
- Variables **must** be declared before they are used
- A variables value may change

```
age = 21; // Compiler error
```

```
int age;
```

```
age = 21;
```

Declaring and Initializing Variables

Declaring Variables

```
VariableType VariableName;
```

```
int age;  
double rate;  
string name;  
  
Account franks_account;  
Person james;
```

Declaring and Initializing Variables

Naming Variables

- Can contain letters, numbers, and underscores
- Must begin with a letter or underscore (_)
 - cannot begin with a number
- Cannot use C++ reserved keywords
- Cannot redeclare a name in the same scope
 - remember that C++ is case sensitive

Declaring and Initializing Variables

Naming Variables

Legal	Illegal
Age	int
age	\$age
_age	2014_age
My_age	My age
your_age_in_2014	Age+1
INT	cout
Int	return

Declaring and Initializing Variables

Naming Variables – Style and Best Practices

- Be consistent with your naming conventions
 - myVariableName vs. my_variable_name
 - avoid beginning names with underscores
- Use meaningful names
 - not too long and not too short
- Never use variables before initializing them
- Declare variables close to when you need them in your code

Declaring and Initializing Variables

Initializing Variables

```
int age; // uninitialized
```

```
int age = 21; // C-like initialization
```

```
int age (21); // Constructor initialization
```

```
int age {21}; // C++11 list initialization syntax
```

C++ Primitive Data Types

- Fundamental data types implemented directly by the C++ language
- Character types
- Integer types
 - signed and unsigned
- Floating-point types
- Boolean type
- Size and precision is often compiler-dependent
 - `#include <climits>`

C++ Primitive Data Types

Type sizes

- Expressed in bits
- The more bits the more values that can be represented
- The more bits the more storage required

Size (in bits)	Representable Values	
8	256	2^8
16	65,536	2^{16}
32	4,294,967,296	2^{32}
64	18,446,744,073,709,551,615	2^{64}

C++ Primitive Data Types

Character Types

- Used to represent single characters, 'A', 'X', '@'
- Wider types are used to represent wide character sets

Type Name	Size / Precision
char	Exactly one byte. At least 8 bits.
char16_t	At least 16 bits.
char32_t	At least 32 bits.
wchar_t	Can represent the largest available character set.

C++ Primitive Data Types

Integer Types

- Used to represent whole numbers
- Signed and unsigned versions

C++ Primitive Data Types

Integer Types

Type Name	Size / Precision
<i>signed short int</i>	At least 16 bits.
<i>signed int</i>	At least 16 bits.
<i>signed long int</i>	At least 32 bits.
<i>signed long long int</i>	At least 64 bits

Type Name	Size / Precision
<i>unsigned short int</i>	At least 16 bits.
<i>unsigned int</i>	At least 16 bits.
<i>unsigned long int</i>	At least 32 bits.
<i>unsigned long long int</i>	At least 64 bits

C++ Primitive Data Types

Floating-point Type

- Used to represent non-integer numbers
- Represented by mantissa and exponent (scientific notation)
- Precision is the number of digits in the mantissa
- Precision and size are compiler dependent

Type Name	Size / Typical Precision	Typical Range
float	/ 7 decimal digits	1.2×10^{-38} to 3.4×10^{38}
double	No less than float / 15 decimal digits	2.2×10^{-308} to 1.8×10^{308}
long double	No less than double / 19 decimal digits	3.3×10^{-4932} to 1.2×10^{4932}

C++ Primitive Data Types

Boolean Type

- Used to represent true and false
- Zero is false.
- Non-zero is true.

Type Name	Size / Precision
bool	Usually 8 bits true or false (C++ keywords)

Using the sizeof operator

- The sizeof operator
 - determines the size in bytes of a type or variable
- Examples:

sizeof(int)

sizeof(double)

sizeof(some_variable)

sizeof some_variable

Using the sizeof operator

<climits> and <cfloat>

- The `climits` and `cfloat` include files contain size and precision information about your implementation of C++

INT_MAX
INT_MIN
LONG_MIN
LONG_MAX
FLT_MIN
FLT_MAX
• • •

What is a constant?

- Like C++ variables
 - Have names
 - Occupy storage
 - Are usually typed

However, their value cannot change once declared!

Types of constants in C++

- Literal constants
- Declared constants
 - `const` keyword
- Constant expressions
 - `constexpr` keyword
- Enumerated constants
 - `enum` keyword
- Defined constants
 - `#define`

Types of constants in C++

Literal constants

- The most obvious kind of constant

x = 12;

y = 1.56;

name = "Frank";

middle_initial = 'J';

Types of constants in C++

Literal constants

- Integer Literal Constants

12 - an integer

12U - an unsigned integer

12L - a long integer

12LL - a long long integer

Types of constants in C++

Literal constants

- Floating-point Literal Constants

12.1 - a double

12.1F - a float

12.1L - a long double

Types of constants in C++

Literal constants

- Character Literal Constants (escape codes)

\n - newline

\r - return

\t - tab

\b - backspace

\' - single quote

\\" - double quote

\\" - backslash

```
cout << "Hello\tthere\nmy friend\n";  
Hello    there  
my friend
```

Types of constants in C++

Declared constants

- Constants declared using the `const` keyword

```
const double pi {3.1415926};
```

```
const int months_in_year {12};
```

```
pi = 2.5; // Compiler error
```

Types of constants in C++

Defined constants

- Constants declared using the `const` keyword

```
#define pi 3.1415926
```

Don't use defined constants in Modern C++

Section Overview

Arrays and Vectors

- Arrays
 - What they are
 - Why we use arrays
 - Declaration and initialization
 - Accessing array elements
- Multi-dimensional arrays
- Vectors
 - What they are
 - Advantages vs. arrays
 - Declaration and initialization

Arrays

What is an array?

- Compound data type or data structure
 - Collection of elements
 - All elements are of the same type
 - Each element can be accessed directly

Arrays

Why do we need arrays?

```
int test_score_1 { 0 } ;  
int test_score_2 { 0 } ;  
int test_score_3 { 0 } ;
```

Arrays

Why do we need arrays?

```
int test_score_1 { 0 } ;  
int test_score_2 { 0 } ;  
int test_score_3 { 0 } ;  
int test_score_4 { 0 } ;  
int test_score_5 { 0 } ;  
.  
.  
. . .  
int test_score_100 { 0 } ;
```

Arrays

Characteristics

- Fixed size
- Elements are all the same type
- Stored contiguously in memory
- Individual elements can be accessed by their position or index
- First element is at index 0
- Last element is at index size-1
- No checking to see if you are out of bounds
- Always initialize arrays
- Very efficient
- Iteration (looping) is often used to process

test_scores	
100	[0]
95	[1]
87	[2]
80	[3]
100	[4]
83	[5]
89	[6]
92	[7]
100	[8]
95	[9]

Arrays

Declaring

```
Element_Type array_name [constant number of elements];
```

```
int test_scores [5];  
  
int high_score_per_level [10];  
  
const int days_in_year { 365};  
double hi_temperatures [days_in_year];
```

Arrays

Initialization

```
Element_Type array_name [number of elements] {init list}
```

```
int test_scores [5] {100,95,99,87,88};  
  
int high_score_per_level [10] {3,5};           // init to 3,5 and remaining to 0  
  
const double days_in_year {365};  
double hi_temperatures [days_in_year] {0}; // init all to zero  
  
int another_array [] {1,2,3,4,5};           // size automatically calculated
```

Arrays

Accessing array elements

```
array_name [element_index]  
  
test_scores [1]
```

```
int test_scores [5] {100,95,99,87,88};  
  
cout << "First score at index 0: " << test_scores[0] << endl;  
cout << "Second score at index 1: " << test_scores[1] << endl;  
cout << "Third score at index 2: " << test_scores[2] << endl;  
cout << "Fourth score at index 3: " << test_scores[3] << endl;  
cout << "Fifth score at index 4: " << test_scores[4] << endl;
```

Arrays

Changing the contents of array elements

array_name **[element_index]**

```
int test_scores [5] {100,95,99,87,88};

cin >> test_scores[0];
cin >> test_scores[1];
cin >> test_scores[2];
cin >> test_scores[3];
cin >> test_scores[4];

test_scores[0] = 90;           // assignment statement
```

Arrays

How does it work?

- The name of the array represent the location of the first element in the array (index 0)
- The [index] represents the offset from the beginning of the array
- C++ simply performs a calculation to find the correct element
- Remember – no bounds checking!

Arrays

Declaring multi-dimensional arrays

```
Element_Type array_name [dim1_size][dim2_size]
```

```
int movie_rating [3][4];
```

Arrays

Multi-dimensional arrays

```
const int rows {3};  
const int cols {4};  
int movie_rating [rows][cols];
```

		movie (second index)			
		0	1	2	3
reviewer (first index)	0	0	4	3	5
	1	2	3	3	5
	2	1	4	4	5

Arrays

Accessing array elements in multi-dimensional arrays

```
cin >> movie_rating [1][2];  
cout << movie_rating [1][2];
```

reviewer
(first index)

movie
(second index)

	0	1	2	3
0	0	4	3	5
1	2	3	5	5
2	1	4	4	5

Arrays

Initializing multi-dimensional arrays

```
int movie_rating [3][4]
{
    { 0, 4, 3, 5 },
    { 2, 3, 3, 5 },
    { 1, 4, 4, 5 }
};
```

	0	1	2	3
0	0	4	3	5
1	2	3	3	5
2	1	4	4	5

Vectors

- Suppose we want to store test scores for my school
- I have no way of knowing how many students will register next year
- Options:
 - Pick a size that you are not likely to exceed and use static arrays
 - Use a dynamic array such as vector

Vectors

What is a vector?

- Container in the C++ Standard Template Library
- An array that can grow and shrink in size at execution time
- Provides similar semantics and syntax as arrays
- Very efficient
- Can provide bounds checking
- Can use lots of cool functions like sort, reverse, find, and more.

Vectors

Declaring

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

vector <char> vowels;

vector <int> test_scores;
```

Vectors

Declaring

```
vector <char> vowels (5);
```

```
vector <int> test_scores (10);
```

Vectors

Initializing

```
vector <char> vowels {'a' , 'e' , 'i' , 'o' , 'u' };
```

```
vector <int> test_scores {100, 98, 89, 85, 93};
```

```
vector <double> hi_temperatures (365, 80.0);
```

Vectors

Characteristics

- Dynamic size
- Elements are all the same type
- Stored contiguously in memory
- Individual elements can be accessed by their position or index
- First element is at index 0
- Last element is at index size-1
- [] - no checking to see if you are out of bounds
- Provides many useful function that do bounds check
- Elements initialized to zero
- Very efficient
- Iteration (looping) is often used to process

Vectors

Accessing vector elements – array syntax

vector_name **[element_index]**

test_scores **[1]**

```
vector <int> test_scores {100,95,99,87,88};

cout << "First score at index 0: " << test_scores[0] << endl;
cout << "Second score at index 1: " << test_scores[1] << endl;
cout << "Third score at index 2: " << test_scores[2] << endl;
cout << "Fourth score at index 3: " << test_scores[3] << endl;
cout << "Fifth score at index 4: " << test_scores[4] << endl;
```

Vectors

Accessing vector elements – vector syntax

```
vector_name.at(element_index)
```

```
test_scores.at(1)
```

```
vector <int> test_scores {100,95,99,87,88};  
  
cout << "First score at index 0: " << test_scores.at(0) << endl;  
cout << "Second score at index 1: " << test_scores.at(1) << endl;  
cout << "Third score at index 2: " << test_scores.at(2) << endl;  
cout << "Fourth score at index 3: " << test_scores.at(3) << endl;  
cout << "Fifth score at index 4: " << test_scores.at(4) << endl;
```

Vectors

Changing the contents of vector elements – vector syntax

```
vector_name.at(element_index)
```

```
vector <int> test_scores {100,95,99,87,88};  
  
cin >> test_scores.at(0);  
cin >> test_scores.at(1);  
cin >> test_scores.at(2);  
cin >> test_scores.at(3);  
cin >> test_scores.at(4);  
  
test_scores.at(0) = 90;           // assignment statement
```

Vectors

So, when do they grow as needed?

```
vector_name.push_back(element)
```

```
vector <int> test_scores {100, 95, 99}; // size is 3  
  
test_scores.push_back(80); // 100, 95, 99, 80  
test_scores.push_back(90); // 100, 95, 99, 80, 90
```

Vector will automatically allocate the required space!

Vectors

What if you are out of bounds?

- Arrays never do bounds checking
- Many vector methods provide bounds checking
- An exception and error message is generated

```
vector <int> test_scores { 100, 95 };

cin >> test_scores.at(5);

terminate called after throwing an instance of 'std::out_of_range'
what(): vector::_M_range_check: __n (which is 5) >= this->size() (which is 2)
This application has requested the Runtime to terminate it in an unusual
way.
Please contact the application's support team for more information.
```

Section Overview

Expressions, Statements and Operators

- Expressions
- Statements and block statements
- Operators
 - Assignment
 - Arithmetic
 - Increment and decrement
 - Equality
 - Relational
 - Logical
 - Compound assignment
 - Precedence

Expressions and Statements

Expressions

An expression is:

- The most basic building block of a program
- “a sequence of operators and operands that specifies a computation”
- Computes a value from a number of operands
- There is much, much more to expressions – not necessary at this level

Expressions and Statements

Expressions - examples

```
34          // literal  
  
favorite_number      // variable  
  
1.5 + 2.8          // addition  
  
2 * 5              // multiplication  
  
a > b              // relational  
  
a = b              // assignment
```

Expressions and Statements

Statements

A statement is:

- A complete line of code that performs some action
- Usually terminated with a semi-colon
- Usually contain expressions
- C++ has many types of statements
 - expression, null, compound, selection, iteration,
 - declaration, jump, try blocks, and others.

Expressions and Statements

Statements - examples

```
int x;                                // declaration  
  
favorite_number = 12;      // assignment  
  
1.5 + 2.8;                            // expression  
  
x = 2 * 5;                             // assignment  
  
if (a > b) cout << "a is greater than b";    // if  
  
;
```

Using Operators

- C++ has a rich set of operators
 - unary, binary, ternary
- Common operators can be grouped as follows:
 - assignment
 - arithmetic
 - increment/decrement
 - relational
 - logical
 - member access
 - other

Assignment Operator (=)

lhs = rhs

- rhs is an expression that is evaluated to a value
- The value of the rhs is stored to the lhs
- The value of the rhs must be type compatible with the lhs
- The lhs must be assignable
- Assignment expression is evaluated to what was just assigned
- More than one variable can be assigned in a single statement

Mixed Type Expressions

- C++ operations occur on same type operands
- If operands are of different types, C++ will convert one
- Important! since it could affect calculation results
- C++ will attempt to automatically convert types (coercion).

If it can't, a compiler error will occur

Mixed Type Expressions

Conversions

- Higher vs. Lower types are based on the size of the values the type can hold
 - long double, double, float, unsigned long, long, unsigned int, int
 - short and char types are always converted to int
- Type Coercion: conversion of one operand to another data type
- Promotion: conversion to a higher type
 - Used in mathematical expressions
- Demotion: conversion to a lower type
 - Used with assignment to lower type

Mixed Type Expressions

Examples

- lower **op** higher **the lower is promoted to a higher**
2 * 5.2
2 is promoted to 2.0
- lower = higher; **the higher is demoted to a lower**
int num {0};
num = 100.2;

Mixed Type Expressions

Explicit Type Casting – static_cast<type>

```
int total_amount {100};  
int total_number {8};  
double average {0.0};  
  
average = total_amount / total_number;  
cout << average << endl;           // displays 12  
  
average = static_cast<double>(total_amount) / total_number;  
cout << average << endl;           // displays 12.5
```

Testing for Equality

The == and != operators

- Compares the values of 2 expressions
- Evaluates to a Boolean (True or False, 1 or 0)
- Commonly used in control flow statements

expr1 == expr2

expr1 != expr2

100 == 200

num1 != num2

Testing for Equality

The == and != operators

```
bool result {false};  
  
result = (100 == 50+50);  
  
result = (num1 != num2);  
  
cout << (num1 == num2) << endl;           // 0 or 1  
cout << std::boolalpha;  
cout << (num1 == num2) << endl;           // true or false  
cout << std::noboolalpha;
```

Relational Operators

`expr1 op expr2`

Operator	Meaning
<code>></code>	greater than
<code>>=</code>	greater than or equal to
<code><</code>	less than
<code><=</code>	less than or equal to
<code><=></code>	three-way comparison (C++20)

Logical Operators

Operator	Meaning
not !	negation
and &&	logical and
or 	logical or

Logical Operators

not (!)

expression	not a
a	!a
true	false
false	true

Logical Operators

and (&&)

expression a	expression b	a and b a && b
true	true	true
true	false	false
false	true	false
false	false	false

Logical Operators

or (||)

expression a	expression b	a or b a b
true	true	true
true	false	true
false	true	true
false	false	false

Logical Operators

Precedence

- not has higher precedence than and
- and has higher precedence than or
- not is a unary operator
- and and or are binary operators

Logical Operators

Examples

```
num1 >= 10 && num1 < 20
```

```
num1 <= 10 || num1 >=20
```

```
!is_raining && temperature > 32.0
```

```
is_raining || is_snowing
```

```
temperature > 100 && is_humid || is_raining
```

Logical Operators

Short-Circuit Evaluation

- When evaluating a logical expression C++ stops as soon as the result is known

```
expr1 && expr2 && expr3
```

```
expr1 || expr2 || expr3
```

Compound Assignment

op=

Operator	Example	Meaning
<code>+=</code>	<code>lhs += rhs;</code>	<code>lhs = lhs + (rhs);</code>
<code>-=</code>	<code>lhs -= rhs;</code>	<code>lhs = lhs - (rhs);</code>
<code>*=</code>	<code>lhs *= rhs;</code>	<code>lhs = lhs * (rhs);</code>
<code>/=</code>	<code>lhs /= rhs;</code>	<code>lhs = lhs / (rhs);</code>
<code>%=</code>	<code>lhs %= rhs;</code>	<code>lhs = lhs % (rhs);</code>
<code>>>=</code>	<code>lhs >>= rhs;</code>	<code>lhs = lhs >> (rhs);</code>
<code><<=</code>	<code>lhs <<= rhs;</code>	<code>lhs = lhs << (rhs);</code>
<code>&=</code>	<code>lhs &= rhs;</code>	<code>lhs = lhs & (rhs);</code>
<code>^=</code>	<code>lhs ^= rhs;</code>	<code>lhs = lhs ^ (rhs);</code>
<code> =</code>	<code>lhs = rhs;</code>	<code>lhs = lhs (rhs);</code>

Logical Operators

Examples

```
lhs op= rhs; // lhs = lhs op (rhs);  
  
a += 1;          // a = a + 1;  
a /= 5;          // a = a / 5;  
a *= b + c;     // a = a * (b + c);  
  
cost += items * tax;  
//cost = cost + (items * tax);
```

Operator Precedence (not a complete list)

Higher to lower

Operator	Associativity
[] -> . ()	left to right
++ -- not -(unary) * (de-ref) & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
= op= ?:	right to left

Operator Precedence

What is associativity?

- Use precedence rules when adjacent operators are different

expr1 **op1** expr2 **op2** expr3 // precedence

- Use associativity rules when adjacent operators have the same precedence

expr1 **op1** expr2 **op1** expr3 // associativity

- Use parenthesis to absolutely remove any doubt

Operator Precedence

Example

```
result = num1 + num2 * num3;  
result = ( num1 + (num2 * num3) );
```

```
result1 = num1 + num2 - num3;  
result1 = ( (num1 + num2) - num3 );
```

Section Overview

Controlling Program Flow

- Sequence
 - Ordering statements sequentially
- Selection
 - Making decisions
- Iteration
 - Looping or repeating

Selection – Decision Making

- if statement
- if-else statement
- Nested if statements
- switch statement
- Conditional operator ?:

Iteration - Looping

- for loop
- Range-based for loop
- while loop
- do-while loop
- continue and break
- Infinite loops
- Nested loops

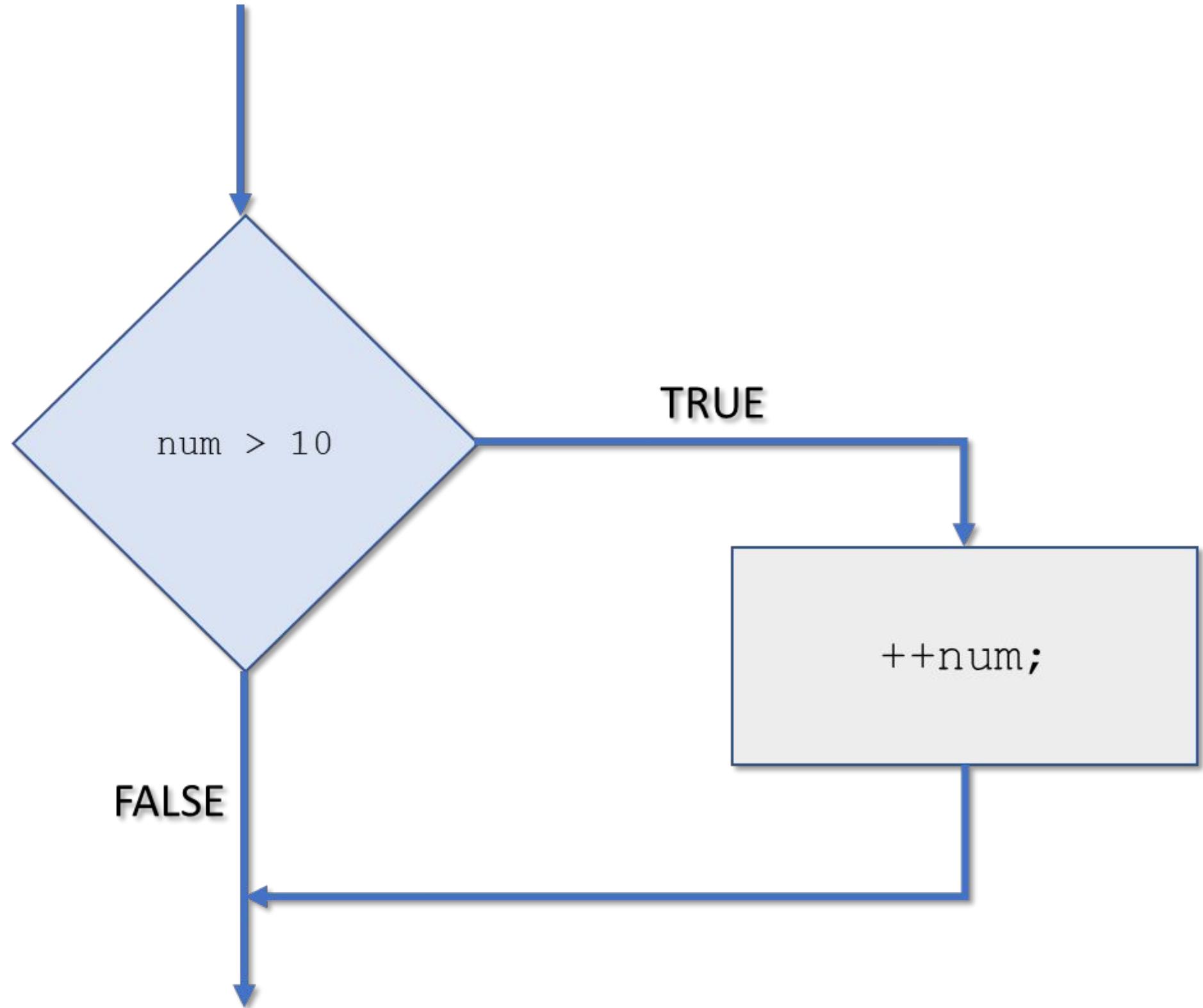
if statement

```
if (expr)  
    statement;
```

- If the expression is true then execute the statement
- If the expression is false then skip the statement

if statement

```
if (num > 10)  
    ++num;
```



if statement

```
if (selection == 'A')
    cout << "You selected A";

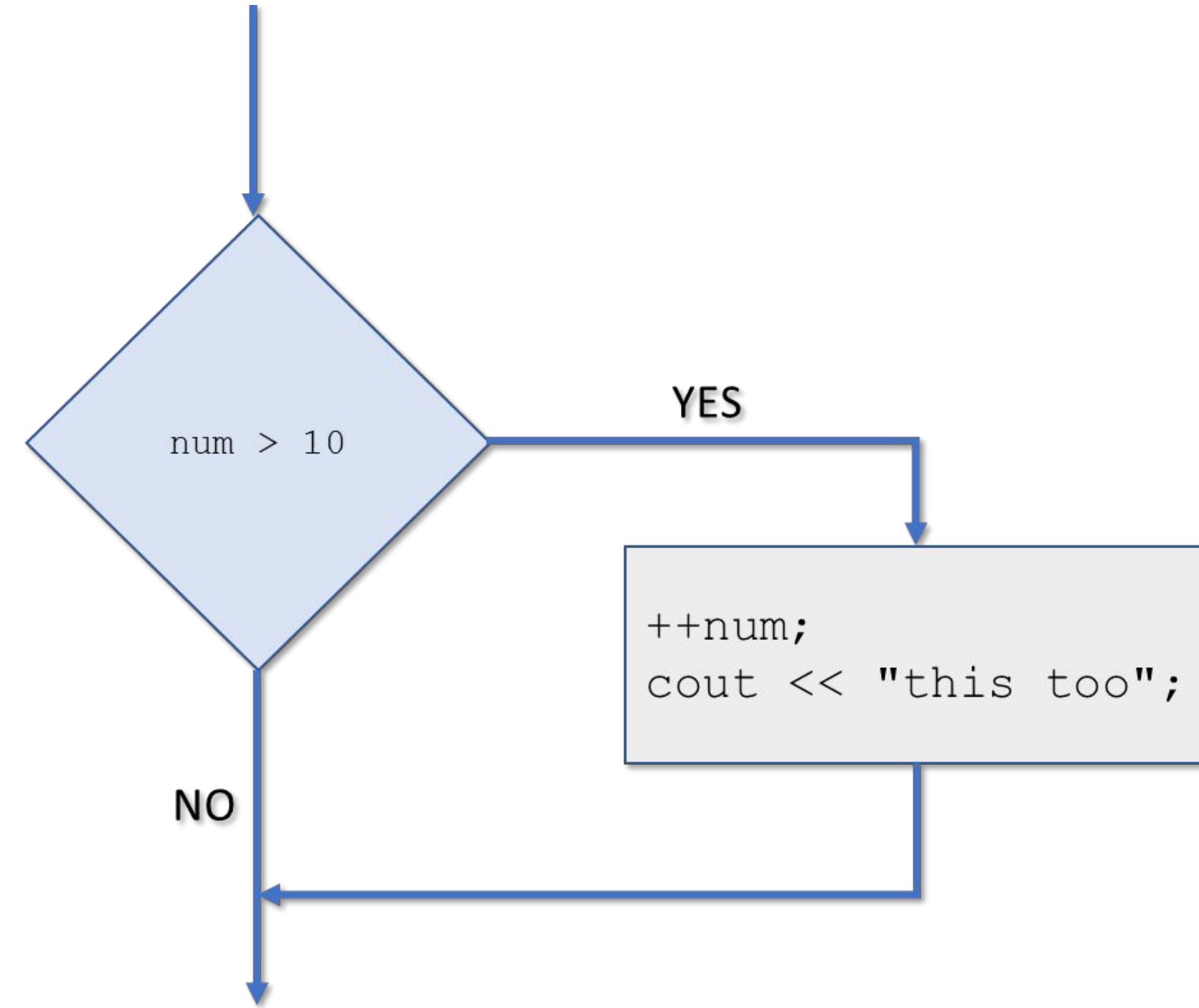
if (num > 10)
    cout << "num is greater than 10";

if (health < 100 && player_healed)
    health = 100;
```

if statement

block statement

```
if(num > 10) {  
    ++num;  
    cout << "this too";  
}
```



Block statement

```
{  
    //variable declarations  
    statement1;  
    statement2;  
    . . .  
}
```

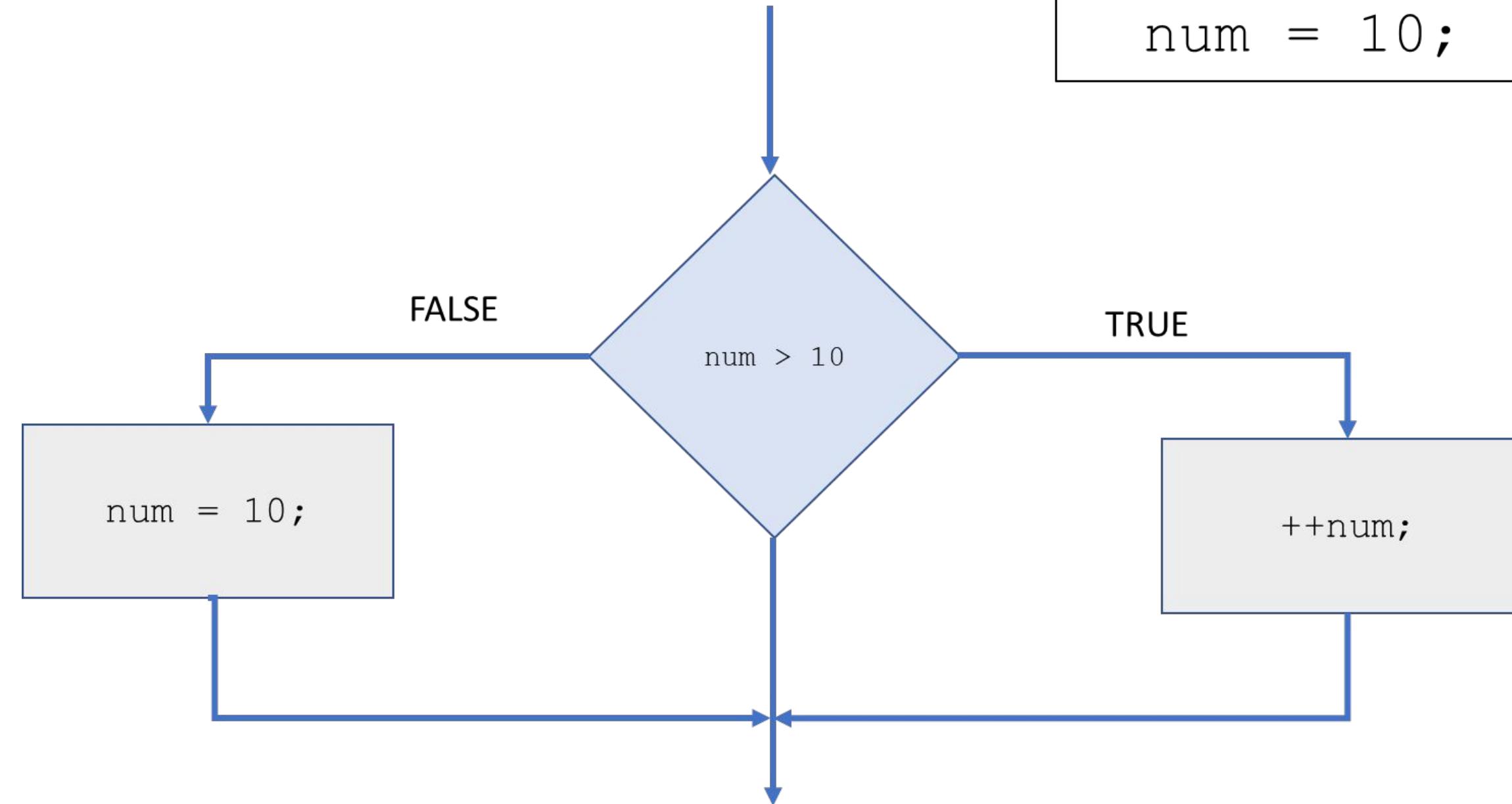
- Create a block of code by including more than one statement in code block {}
- Blocks can also contain variable declarations
- These variables are visible only within the block – local scope

if-else statement

```
if (expr)
    statement1;
else
    statement2;
```

- If the expression is **true** then execute **statement1**
- If the expression is **false** then execute **statement2**

if-else statement



```
if (num > 10)  
    ++num;  
else  
    num = 10;
```

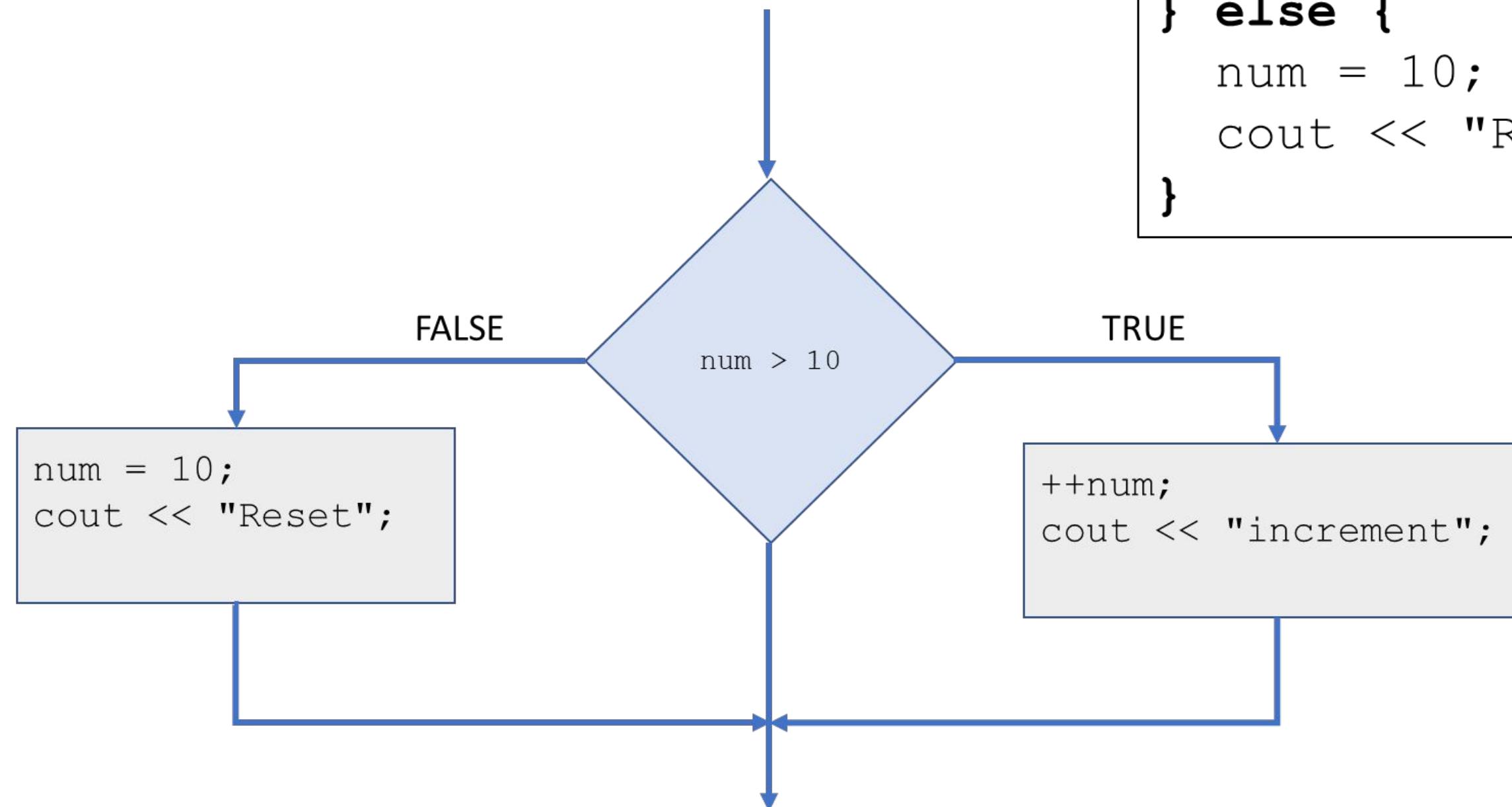
if-else statement

```
if (num > 10)
    cout << "num is greater than 10";
else
    cout << "num is NOT greater than 10";

if (health < 100 && heal_player)
    health = 100;
else
    ++health;
```

if-else statement

block statement



```
if (num > 10) {  
    ++num;  
    cout << "increment";  
} else {  
    num = 10;  
    cout << "Reset";  
}
```

if-else-if construct

block statement

```
if (score > 90)
    cout << "A";
else if (score > 80)
    cout << "B";
else if (score > 70)
    cout << "C";
else if (score > 60)
    cout << "D";
else // all others must be F
    cout << "F";
cout << "Done";
```

Nested if statement

```
if (expr1)
    if (expr2)
        statement1;
else
    statement2;
```

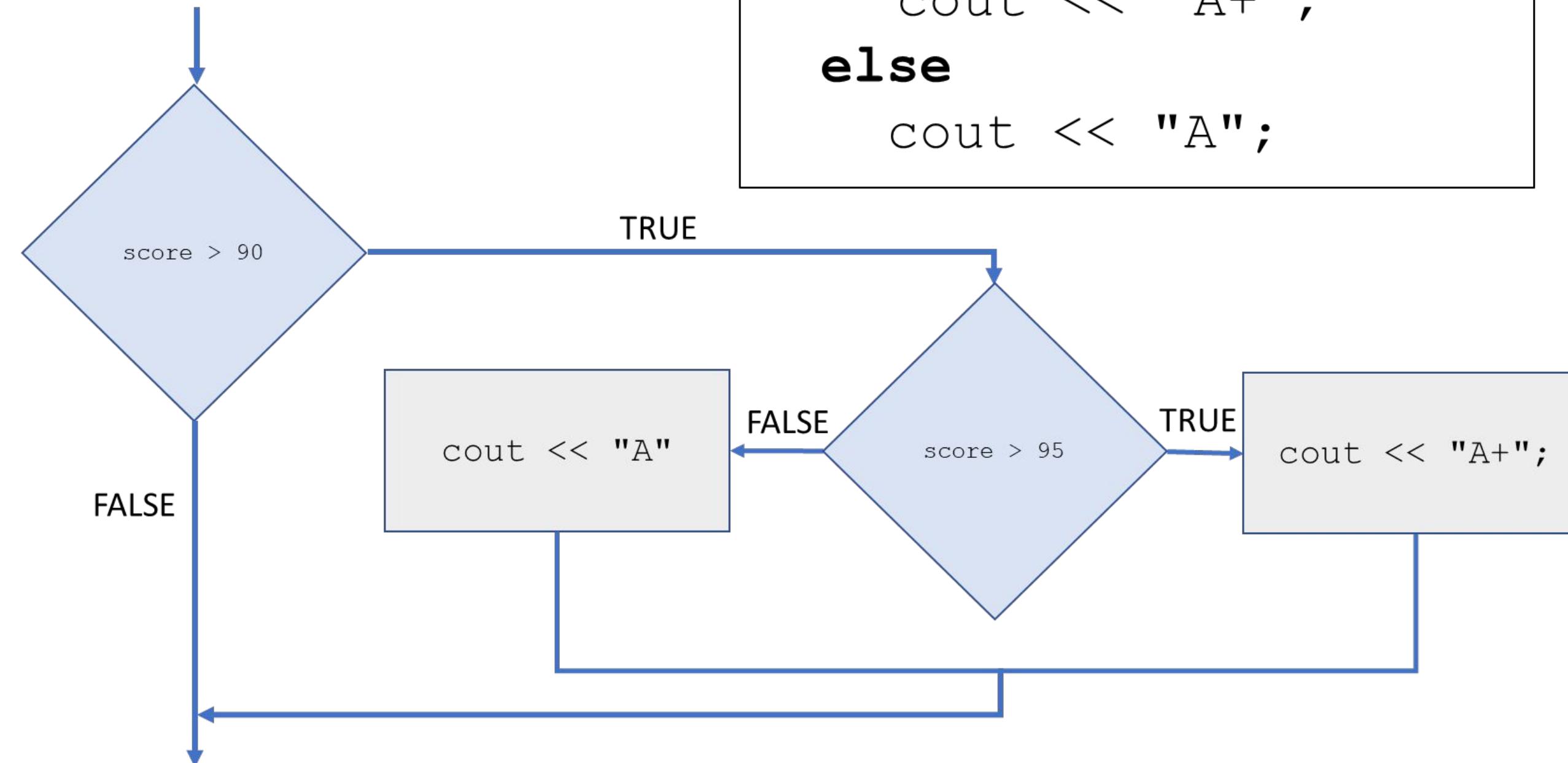
- **if** statement is nested within another
- Allows testing of multiple conditions
- **else** belongs to the closest **if**

Nested if statement

```
if (score > 90)
    if (score > 95)
        cout << "A+";
else
    cout << "A";

else
    cout << "Sorry, No A";
```

Nested if statement



Nested if statement

```
if (score_frank != score_bill) {  
    if (score_frank > score_bill) {  
        cout << "Frank Wins" << endl;  
    } else {  
        cout << "Bill Wins" << endl;  
    }  
} else {  
    cout << "Looks like a tie!" << endl;  
}
```

The switch statement

```
switch (integer_control_expr) {  
    case expression_1: statement_1; break;  
    case expression_2: statement_2; break;  
    case expression_3: statement_3; break;  
    . . .  
    case expression_n: statement_n; break;  
    default: statement_default;  
}
```

The switch statement

example

```
switch (selection) {  
    case '1': cout << "1 selected";  
                break;  
    case '2': cout << "2 selected";  
                break;  
    case '3':  
    case '4': cout << "3 or 4 selected";  
                break;  
    default:   cout << "1,2,3,4 NOT selected";  
}
```

The switch statement

fall-through example

```
switch (selection) {  
    case '1': cout << "1 selected";  
  
    case '2': cout << "2 selected";  
  
    case '3': cout << "3 selected";  
  
    case '4': cout << "4 selected";  
                break;  
    default:   cout << "1,2,3,4 NOT selected";  
}
```

The switch statement

with an enumeration

```
enum Color {  
    red, green, blue  
};  
Color screen_color {green};
```

```
switch (screen_color) {  
    case red:    cout << "red"; break;  
    case green:  cout << "green"; break;  
    case blue:   cout << "blue"; break;  
    default:     cout << "should never execute";  
}
```

The switch statement

- The control expression must evaluate to an integer type
- The case expressions must be constant expressions that evaluate to integer or integers literals
- Once a match occurs all following case sections are executes UNTIL a break is reached the switch complete
- Best practice – provide break statement for each case
- Best practice – default is optional, but should be handled

Conditional Operator

?:

```
(cond_expr) ? expr1 : expr2
```

- cond_expr evaluates to a boolean expression
 - If cond_expr is true then the value of expr1 is returned
 - If cond_expr is false then the value of expr2 is returned
- Similar to if-else construct
- Ternary operator
- Very useful when used inline
- Very easy to abuse!

Conditional Operator

example

```
int a{10}, b{20};  
int score{92};  
int result {};  
  
result = (a > b) ? a : b;  
  
result = (a < b) ? (b-a) : (a-b);  
  
result = (b != 0) ? (a/b) : 0;  
  
cout << ((score > 90) ? "Excellent" : "Good ");
```

Looping

iteration

- The third basic building block of programming
 - sequence, selection, **iteration**
- Iteration or repetition
- Allows the execution of a statement or block of statements repeatedly
- Loops are made up a loop condition and the body which contains the statements to repeat

Looping

Some typical use-cases

Execute a loop:

- a specific number of times
- for each element in a collection
- while a specific condition remains true
- until a specific condition becomes false
- until we reach the end of some input stream
- forever
- many, many more

C++ Looping Constructs

- **for loop**
 - iterate a specific number of times
- Range-based **for loop**
 - one iteration for each element in a range or collection
- **while loop**
 - iterate while a condition remains true
 - stop when the condition becomes false
 - check the condition at the beginning of every iteration
- **do-while loop**
 - iterate while a condition remains true
 - stop when the condition becomes false
 - check the condition at the end of every iteration

for Loop

```
for (initialization ; condition ; increment)
    statement;

for (initialization ; condition ; increment) {
    statement(s);
}
```

for Loop

```
int i {0};  
  
for (i = 1 ; i <= 5 ; ++i)  
    cout << i << endl;
```

```
1  
2  
3  
4  
5
```

for Loop

```
for (int i {1} ; i <= 5 ; ++i)  
    cout << i << endl;
```

```
for (int i = 1 ; i <= 5 ; ++i)  
    cout << i << endl;
```

i = 100; // ERROR i only visible in the loop

for Loop

display even numbers

```
for (int i {1} ; i <= 10 ; ++i) {
    if (i % 2 == 0)
        cout << i << endl;
```

2
4
6
8
10

for Loop

array example

```
int scores [] {100,90,87};

for (int i {0} ; i < 3 ; ++i) {
    cout << scores[i] << endl;
}

for (int i {0} ; i <= 2 ; ++i) {
    cout << scores[i] << endl;
}
100
90
87
```

for Loop

comma operator

```
for (int i {1}, j {5} ; i <= 5 ; ++i, ++j) {
    cout << i << " * " << j << " : " << (i * j) << endl;
}
```

```
1 * 5 : 5
2 * 6 : 12
3 * 7 : 21
4 * 8 : 32
5 * 9 : 45
```

for Loop

some other details...

- The basic for loop is very clear and concise
- Since the for loop's expressions are all optional, it is possible to have
 - no initialization
 - no test
 - no increment

```
for (;;) {  
    cout << "Endless loop" << endl;  
}
```

Range-based for Loop

Introduced in C++11

```
for (var_type var_name: sequence)
    statement; // can use var_name
```

```
for (var_type var_name: sequence) {
    statements; // can use var_name
}
```

Range-based for Loop

```
int scores [] {100, 90, 97};  
  
for (int score : scores)  
    cout << score << endl;
```

```
100  
90  
97
```

Range-based for Loop

auto

```
int scores [] {100, 90, 97};  
  
for (auto score : scores)  
    cout << score << endl;
```

```
100  
90  
97
```

Range-based for Loop

vector

```
vector<double> temps {87.2, 77.1, 80.0, 72.5};

double average_temp {};
double running_sum {};

for (auto temp: temps)
    running_sum += temp;

average_temp = running_sum / temps.size();
```

Range-based for Loop

initializer list

```
double average_temp {};  
double running_sum {};  
int size {0};  
  
for (auto temp: {60.2, 80.1, 90.0, 78.2}) {  
    running_sum += temp;  
    ++size;  
}  
average_temp = running_sum / size;
```

Range-based for Loop

string

```
for (auto c: "Frank")
    cout << c << endl;
```

F
r
a
n
k

while Loop

```
while (expression)
    statement;

while (expression) {
    statement(s);
}
```

while Loop

```
int i {1};

while (i <= 5) {
    cout << i << endl;
    ++i; // important!
}
```

```
1
2
3
4
5
```

while Loop

even numbers

```
int i {1};

while (i <= 10) {
    if (i % 2 == 0)
        cout << i << endl;
    ++i;
}
```

```
2  
4  
6  
8  
10
```

while Loop

array example

```
int scores [] {100, 90, 87};  
int i {0};  
  
while (i < 3) {  
    cout << scores[i] << endl;  
    ++i;  
}
```

```
100  
90  
87
```

while Loop

input validation

```
int number { };

cout << "Enter an integer less than 100: ";
cin >> number;

while (number >= 100) { // !(number < 100)
    cout << "Enter an integer less than 100";
    cin >> number;
}

cout << "Thanks" << endl;
```

while Loop

input validation

```
int number {};  
  
cout << "Enter an integer between 1 and 5: ";  
cin >> number;  
  
while (number <= 1 || number >= 5) {  
    cout << "Enter an integer between 1 and 5: ";  
    cin >> number;  
}  
  
cout << "Thanks" << endl;
```

while Loop

input validation - boolean flag

```
bool done {false};  
int number {0};  
  
while (!done) {  
    cout << "Enter an integer between 1 and 5: ";  
    cin >> number;  
    if (number <=1 || number >=5)  
        cout << "Out of range, try again" << endl;  
    else {  
        cout << "Thanks!" << endl;  
        done = true;  
    }  
}
```

do-while Loop

```
do {  
    statements;  
} while (expression);
```

do-while Loop

input validation

```
int number { };
do {

    cout << "Enter an integer between 1 and 5: ";
    cin >> number;

} while (number <= 1 || number >= 5);

cout << "Thanks" << endl;
```

do-while Loop

area calculation with calculate another

```
char selection { };

do {
    double width {}, height {};
    cout << "Enter width and height separated by a space :";
    cin >> width >> height;

    double area {width * height };
    cout << "The area is " << area << endl;

    cout << "Calculate another? (Y/N) : ";
    cin >> selection;
} while (selection == 'Y' || selection == 'y');
cout << "Thanks!" << endl;
```

continue and break statements

- continue
 - no further statements in the body of the loop are executed
 - control immediately goes directly to the beginning of the loop for the next iteration
- break
 - no further statements in the body of the loop are executed
 - loop is immediately terminated
 - Control immediately goes to the statement following the loop construct

continue and break statements

```
vector<int> values {1,2,-1,3,-1,-99,7,8,10};  
  
for (auto val: values) {  
  
    if (val == -99)  
  
        break;  
  
    else if (val == -1)  
  
        continue;  
  
    else  
  
        cout << val << endl;  
  
}  
  
1  
2  
3
```

Infinite Loops

- Loops whose condition expression always evaluate to true
- Usually this is unintended and a programmer error
- Sometimes programmers use infinite loops and include and break statements in the body to control them
- Sometimes infinite loops are exactly what we need
 - Event loop in an event-driven program
 - Operating system

Infinite for Loops

```
for (;;)
    cout << "This will print forever" << endl;
```

Infinite while Loops

```
while (true)
    cout << "This will print forever" << endl;
```

Infinite do-while Loops

```
do {  
    cout << "This will print forever" << endl;  
} while (true);
```

Infinite while Loops

example

```
while (true) {  
    char again {};  
    cout << "Do you want to loop again? (Y/N) : ";  
    cin >> again;  
  
    if (again == 'N' || again == 'n')  
        break;  
}
```

Nested Loops

- Loop nested within another loop
- Can be many as many levels deep as the program needs
- Very useful with multi-dimensional data structures
- Outer loop vs. Inner loop

Nested Loops

```
for (outer_val {1}; outer_val <= 2 ; ++outer_val)
    for (inner_val {1}; inner_val <= 3; ++inner_val)
        cout << outer_val << "," << inner_val << endl;
```

1, 1

1, 2

1, 3

outer_val, inner_val

2, 1

Note: inner loop loops “faster”

2, 2

2, 3

Nested Loops

Multiplication Table

```
for (int num1 {1}; num1 <=10 ; ++num1) { // outer
    for (int num2 {1}; num2 <=10; ++num2) { // inner
        cout << num1 << " * " << num2
        << " = " << num1 * num2 << endl;
    }
    cout << "-----" << endl;
}
```

Displays 10 x 10 Multiplication Table

Nested Loops

2D Arrays – set all elements to 1000

```
int grid[5][3] { };

for (int row {0}; row < 5; ++row) {
    for (int col {0}; col < 3; ++col) {
        grid[row][col] = 1000;
    }
}
```

Nested Loops

2D Arrays – display elements

```
for (int row {0}; row < 5; ++row) {
    for (int col {0}; col < 3; ++col) {
        cout << grid[row][col] << " ";
    }
    cout << endl;
}
```

Nested Loops

2D Vector - display elements

```
vector<vector<int>> vector_2d
{
    {1, 2, 3},
    {10, 20, 30, 40},
    {100, 200, 300, 400, 500}
};

for (auto vec: vector_2d) {
    for (auto val: vec) {
        cout << val << " ";
    }
    cout << endl;
}
```

Output

```
1 2 3
10 20 30 40
100 200 300 400 500
```

Section Overview

Characters and Strings

- Character functions
- C-style Strings
- Working with C-style Strings
- C++ Strings
- Working with C++ Strings

Character Functions

```
#include <cctype>
```

```
#include <cctype>
```

```
function_name(char)
```

- Functions for testing characters
- Functions for converting character case

Character Functions

Testing characters

isalpha (c)	True if c is a letter
isalnum (c)	True if c is a letter or digit
isdigit (c)	True if c is a digit
islower (c)	True if c is lowercase letter
isprint (c)	True if c is a printable character
ispunct (c)	True if c is a punctuation character
isupper (c)	True if c is an uppercase letter
isspace (c)	True if c is whitespace

Character Functions

Converting characters

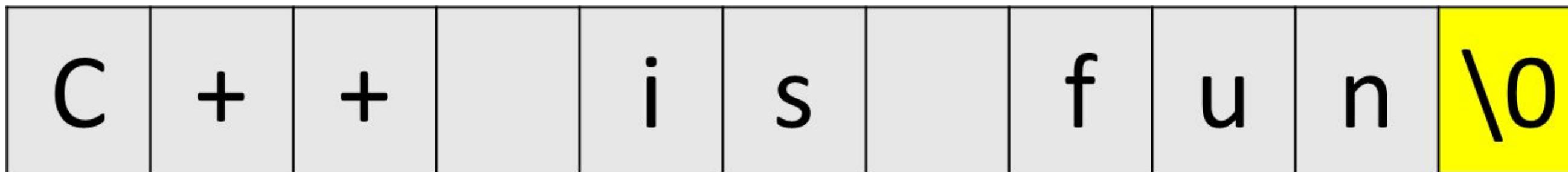
tolower (c)	returns lowercase of c
toupper (c)	returns uppercase of c

C-style Strings

- Sequence of characters
 - contiguous in memory
 - implemented as an array of characters
 - terminated by a null character (null)
 - null – character with a value of zero
 - Referred to as zero or null terminated strings
- String literal
 - sequence of characters in double quotes, e.g. "Frank"
 - constant
 - terminated with null character

C-style Strings

"C++ is fun"



C-style Strings

declaring variables

```
char my_name[] {"Frank"};
```

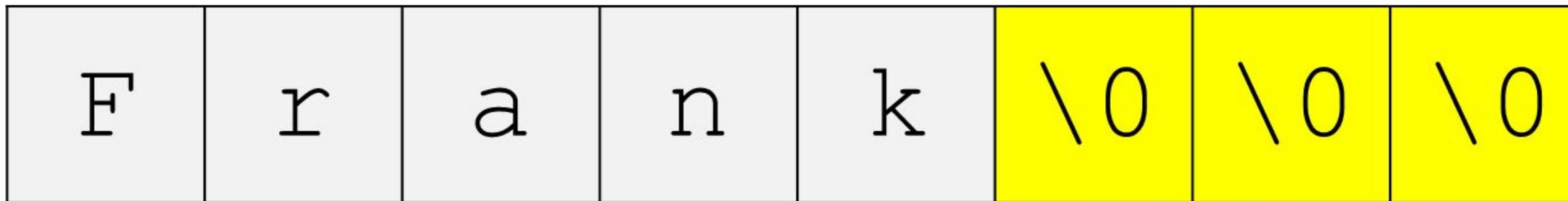
F	r	a	n	k	\0
---	---	---	---	---	----

```
my_name[5] = 'y'; // Problem
```

C-style Strings

declaring variables

```
char my_name[8] {"Frank"};
```



```
my_name[5] = 'y'; // OK
```

C-style Strings

declaring variables

```
char my_name[8];
```

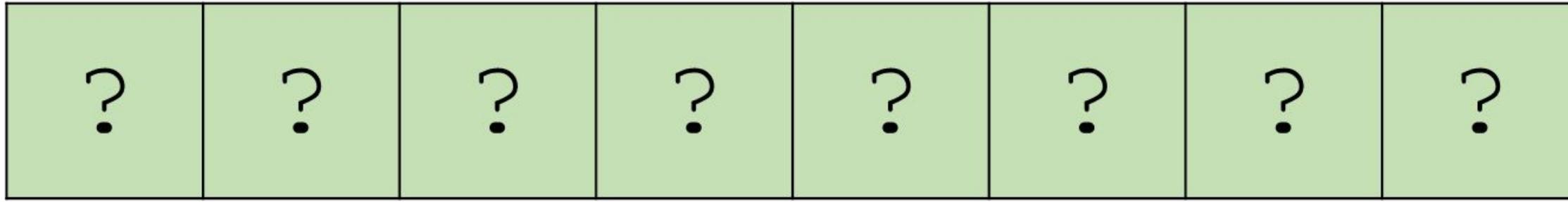


Diagram illustrating the declaration of a character array `my_name[8]`. It shows eight adjacent green rectangular boxes, each containing a question mark ('?'), representing the initial state of the memory allocated for the string.

```
my_name = "Frank"; // Error
```

```
strcpy(my_name, "Frank"); // OK
```

#include <cstring>

Functions that work with C-style Strings

- Copying
- Concatenation
- Comparison
- Searching
- and others

#include <cstring>

A few examples

```
char str[80];  
  
strcpy(str, "Hello "); // copy  
  
strcat(str, "there"); // concatenate  
  
cout << strlen(str); // 11  
  
strcmp(str, "Another"); // > 0
```

#include <cstdlib>

General purpose functions

- Includes functions to convert C-style Strings to
- integer
- float
- long
- etc.

C++ Strings

- std::string is a Class in the Standard Template Library
 - #include <string>
 - std namespace
 - contiguous in memory
 - dynamic size
 - work with input and output streams
 - lots of useful member functions
 - our familiar operators can be used (+, =, <, <=, >, >=, +=, ==, !=, []...)
 - can be easily converted to C-style Strings if needed
 - safer

C++ Strings

Declaring and initializing

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

string s1;                                // Empty
string s2 {"Frank"};                      // Frank
string s3 {s2};                            // Frank
string s4 {"Frank", 3};                    // Fra
string s5 {s3, 0, 2};                      // Fr
string s6 (3, 'X');                       // XXX
```

C++ Strings

Assignment =

```
string s1;  
s1 = "C++ Rocks!";  
  
string s2 {"Hello"};  
s2 = s1;
```

C++ Strings

Concatenation

```
string part1 {"C++"};
string part2 {"is a powerful"};

string sentence;

sentence = part1 + " " + part2 + " language";
// C++ is a powerful language

sentence = "C++" + " is powerful"; // Illegal
```

C++ Strings

Accessing characters [] and at() method

```
string s1;  
string s2 {"Frank"};  
  
cout << s2[0] << endl;      // F  
cout << s2.at(0) << endl; // F  
  
s2[0] = 'f';      // frank  
s2.at(0) = 'p';   // prank
```

C++ Strings

Accessing characters [] and at() method

```
string s1 {"Frank"};  
  
for (char c: s1)  
    cout << c << endl;
```

F
r
a
n
k

C++ Strings

Accessing characters [] and at() method

```
string s1 {"Frank"};  
  
for (int c: s1)  
    cout << c << endl;  
  
70 // F  
114 // r  
97 // a  
110 // n  
107 // k  
0 // null character
```

C++ Strings

Comparing

== != > >= < <=

The objects are compared character by character lexically.

Can compare:

two `std::string` objects

`std::string` object and C-style string literal

`std::string` object and C-style string variable

C++ Strings

Comparing

```
string s1 {"Apple"};
string s2 {"Banana"};
string s3 {"Kiwi"};
string s4 {"apple"};
string s5 {s1};           // Apple

s1 == s5                // True
s1 == s2                // False
s1 != s2                // True
s1 < s2                 // True
s2 > s1                 // True
s4 < s5                 // False
s1 == "Apple";          // True
```

C++ Strings

Substrings - substr()

Extracts a substring from a std::string

object.substr(start_index, length)

```
string s1 {"This is a test"};  
  
cout << s1.substr(0, 4); // This  
cout << s1.substr(5, 2); // is  
cout << s1.substr(10, 4); // test
```

C++ Strings

Searching - find()

Returns the index of a substring in a std::string

object.find(search_string)

```
string s1 {"This is a test"};  
  
cout << s1.find("This"); // 0  
cout << s1.find("is"); // 2  
cout << s1.find("test"); // 10  
cout << s1.find('e'); // 11  
cout << s1.find("is", 4); // 5  
cout << s1.find("XX"); // string::npos
```

C++ Strings

Removing characters - `erase()` and `clear()`

Removes a substring of characters from a `std::string`

`object.erase(start_index, length)`

```
string s1 {"This is a test"};
```

```
cout << s1.erase(0,5); // is a test
```

```
cout << s1.erase(5,4); // is a
```

```
s1.clear(); // empties string s1
```

C++ Strings

Other useful methods

```
string s1 {"Frank"};
```

```
cout << s1.length() << endl; // 5
```

```
s1 += " James";
```

```
cout << s1 << endl; // Frank James
```

Many more...

C++ Strings

Input >> and getline()

Reading std::string from cin

```
string s1;  
cin >> s1;          // Hello there  
                      // Only accepts up to the first space  
cout << s1 << endl; // Hello  
  
getline(cin, s1);   // Read entire line until \n  
cout << s1 << endl; // Hello there  
  
getline(cin, s1, 'x'); // this is x  
cout << s1 << endl; // this is
```

Functions

- Function
 - definition
 - prototype
 - Parameters and pass-by-value
 - return statement
 - default parameter values
 - overloading
 - passing arrays to function
 - pass-by-reference
 - inline functions
 - auto return type
 - recursive functions

What is a function?

- C++ programs
 - C++ Standard Libraries (functions and classes)
 - Third-party libraries (functions and classes)
 - Our own functions and classes
- Functions allow the modularization of a program
 - Separate code into logical self-contained units
 - These units can be reused

What is a function?

```
int main() {  
  
    // read input  
    statement1;  
    statement2;  
    statement3;  
    statement4;  
  
    // process input  
    statement5;  
    statement6;  
    statement7;  
  
    // provide output  
    statement8;  
    statement9;  
    statement10;  
  
    return 0;  
}
```

Modularized Code

```
int main() {  
  
    // read input  
    read_input();  
  
    // process input  
    process_input();  
  
    // provide output  
    provide_output();  
  
    return 0;  
}
```

What is a function?

```
int main() {  
  
    read_input();  
  
    process_input();  
  
    provide_output();  
  
    return 0;  
}
```

```
read_input() {  
    statement1;  
    statement2;  
    statement3;  
    statement4;  
}  
  
process_input() {  
    statement5;  
    statement6;  
    statement7;  
}  
  
provide_output() {  
    statement8;  
    statement9;  
    statement10;  
}
```

What is a function?

Boss/Worker analogy

- Write your code to the function specification
- Understand what the function does
- Understand what information the function needs
- Understand what the function returns
- Understand any errors the function may produce
- Understand any performance constraints

- Don't worry about HOW the function works internally
 - Unless you are the one writing the function!

What is a function?

Example <cmath>

- Common mathematical calculations
- Global functions called as:

```
function_name(argument);  
function_name(argument1, argument2, ...);  
  
cout << sqrt(400.0) << endl; // 20.0  
double result;  
result = pow(2.0, 3.0); // 2.0^3.0
```

What is a function?

User-defined functions

- We can define our own functions
- Here is a preview

```
/* This is a function that expects two integers a and b
   It calculates the sum of a and b and returns it to the caller
   Note that we specify that the function returns an int
*/
int add_numbers(int a, int b)
{
    return a + b;
}

// I can call the function and use the value that is returns

cout << add_numbers(20, 40);
```

What is a function?

User-defined functions

- Return zero if any of the arguments are negative

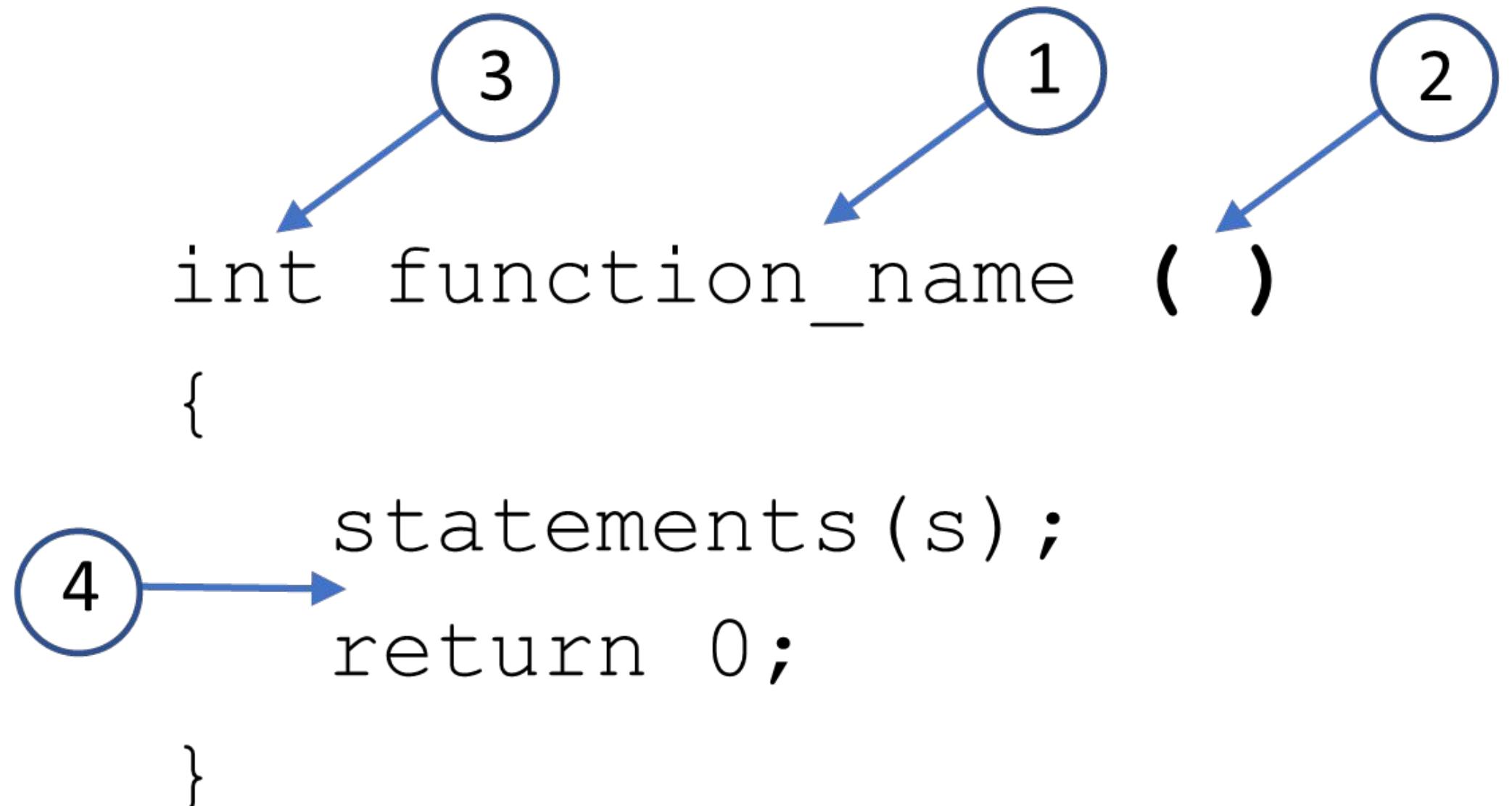
```
/* This is a function that expects two integers a and b
   It calculates the sum of a and b and returns it to the caller
   Only if a or b are non-negative. Otherwise, it returns 0
   Note that we specify that the function returns an int
*/
int add_numbers(int a, int b)
{
    if (a < 0 || b < 0)
        return 0;
    else
        return a + b;
}
```

Defining Functions

- name
 - the name of the function
 - same rules as for variables
 - should be meaningful
 - usually a verb or verb phrase
- parameter list
 - the variables passed into the function
 - their types must be specified
- return type
 - the type of the data that is returned from the function
- body
 - the statements that are executed when the function is called
 - in curly braces {}

Defining Functions

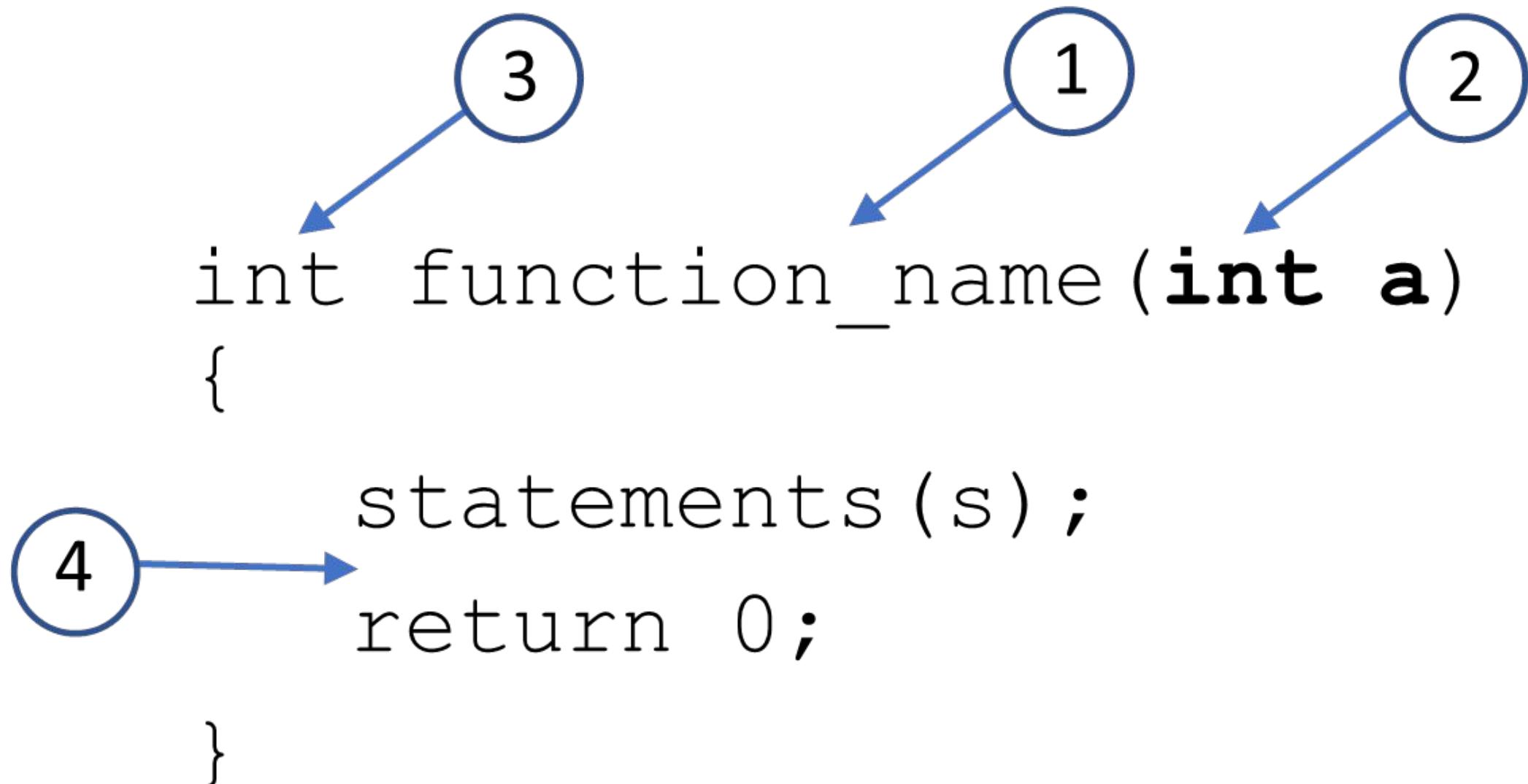
Example with no parameters



1. Name
2. Parameters
3. Return type
4. Body

Defining Functions

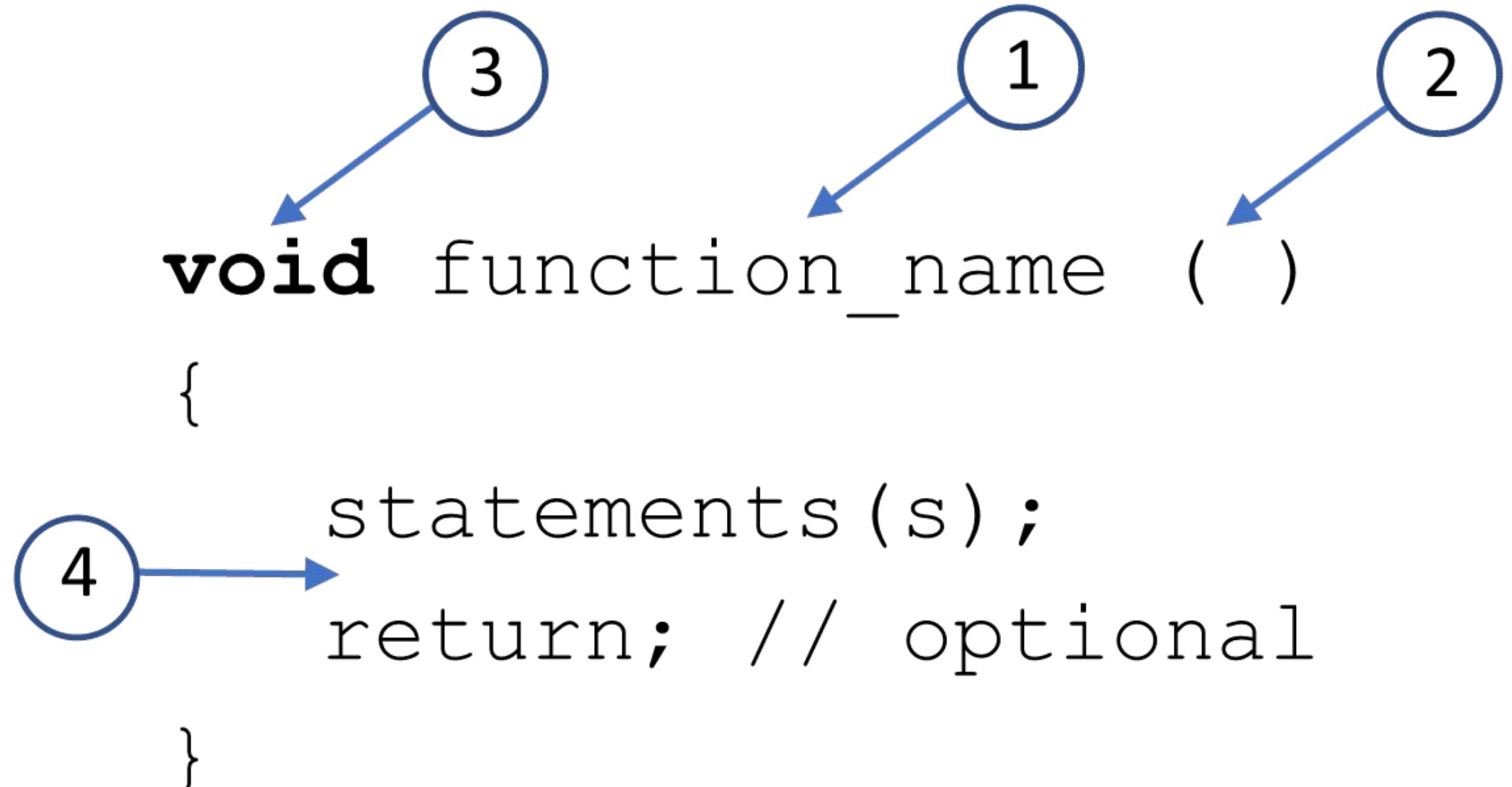
Example with 1 parameter



1. Name
2. Parameters
3. Return type
4. Body

Defining Functions

Example with no return type (void)



1. Name
2. Parameters
3. No return type
4. Body

Defining Functions

Example with multiple parameters

```
void function_name(int a, std::string b)
{
    statements(s);
    return; // optional
}
```

Defining Functions

A function with no return type and no parameters

```
void say_hello () {  
    cout << "Hello" << endl;  
}
```

Calling a function

```
void say_hello () {  
    cout << "Hello" << endl;  
}  
  
int main () {  
    say_hello();  
    return 0;  
}
```

Calling a function

```
void say_hello () {  
    cout << "Hello" << endl;  
}  
  
int main () {  
    for (int i{1} ; i<=10; ++i)  
        say_hello();  
    return 0;  
}
```

Calling a function

```
void say_world () {  
    cout << " World" << endl;  
}
```

```
void say_hello () {  
    cout << "Hello" << endl;  
    say_world();  
}
```

```
int main() {  
    say_hello();  
    return 0;  
}
```

Calling a function

```
void say_world () {  
    cout << " World" << endl;  
    cout << " Bye from say_world" << endl;  
}  
  
void say_hello () {  
    cout << "Hello" << endl;  
    say_world();  
    cout << " Bye from say_hello" << endl;  
}  
  
int main() {  
    say_hello();  
    cout << " Bye from main" << endl;  
    return 0;  
}
```

```
Hello  
World  
Bye from say_world  
Bye from say_hello  
Bye from main
```

Calling functions

- Functions can call other functions
- Compiler must know the function details **BEFORE** it is called!

```
int main() {  
    say_hello(); // called BEFORE it is defined ERROR  
    return 0;  
}  
  
void say_hello ()  
{  
    cout << "Hello" << endl;  
}
```

Function Prototypes

- **The compiler must 'know' about a function before it is used**
 - Define functions before calling them
 - OK for small programs
 - Not a practical solution for larger programs
 - Use function prototypes
 - Tells the compiler what it needs to know without a full function definition
 - Also called forward declarations
 - Placed at the beginning of the program
 - Also used in our own header files (.h) – more about this later

Example

```
int function_name(); // prototype
```

```
int function_name()
{
    statements(s);
    return 0;
}
```

Example

```
int function_name(int); // prototype  
                      // or
```

```
int function_name(int a); // prototype
```

```
int function_name(int a) {  
    statements(s);  
    return 0;  
}
```

Example

```
void function_name(); // prototype
```

```
void function_name()
{
    statements(s);
    return; // optional
}
```

Example

```
void function_name(int a, std::string b);
```

// or

```
void function_name(int, std::string);
```

```
void function_name(int a, std::string b)
```

```
{
```

```
    statements(s);
```

```
    return; // optional
```

```
}
```

A function with no return type and no parameters

```
void say_hello();
```

```
void say_hello() {  
    cout << "Hello" << endl;  
}
```

Calling a function

```
void say_hello();  
  
int main() {  
    say_hello();           // OK  
    say_hello(100);       // Error  
    cout << say_hello(); // Error  
                           // No return value  
    return 0;  
}
```

Example

```
void say_hello(); // prototype
void say_world(); // prototype

int main() {
    say_hello();
    cout << " Bye from main" << endl;
    return 0;
}
```

```
void say_world () {
    cout << " World" << endl;
    cout << " Bye from say_world" << endl;
}

void say_hello () {
    cout << "Hello" << endl;
    say_world();
    cout << " Bye from say_hello" << endl;
}
```

Function Parameters

- When we call a function we can pass in data to that function
- In the function call they are called arguments
- In the function definition they are called parameters
- They must match in number, order, and in type

Example

```
int add_numbers(int, int); // prototype
```

```
int main() {
    int result {0};
    result = add_numbers(100,200); // call
    return 0;
}
```

```
int add_numbers(int a, int b) { // definition
    return a + b;
}
```

Example

```
void say_hello(std::string name) {  
    cout << "Hello " << name << endl;  
}
```

```
say_hello("Frank");
```

```
std::string my_dog {"Buster"};  
say_hello(my_dog);
```

Pass-by-value

- When you pass data into a function it is passed-by-value
 - A copy of the data is passed to the function
 - Whatever changes you make to the parameter in the function does NOT affect the argument that was passed in.
-
- Formal vs. Actual parameters
 - Formal parameters – the parameters defined in the function header
 - Actual parameters – the parameter used in the function call, the arguments

Example

```
void param_test(int formal) { // formal is a copy of actual
    cout << formal << endl; // 50
    formal = 100;           // only changes the local copy
    cout << formal << endl; // 100
}

int main() {
    int actual {50};
    cout << actual << endl; // 50
    param_test(actual);       // pass in 50 to param_test
    cout << actual << endl; // 50 - did not change
    return 0
}
```

Function Return Statement

- If a function returns a value then it must use a `return` statement that returns a value
- If a function does not return a value (`void`) then the `return` statement is optional
- `return` statement can occur anywhere in the body of the function
- `return` statement immediately exits the function
- We can have multiple `return` statements in a function
 - Avoid many `return` statements in a function
- The return value is the result of the function call

Default Argument Values

- When a function is called, all arguments must be supplied
- Sometimes some of the arguments have the same values most of the time
- We can tell the compiler to use default values if the arguments are not supplied
- Default values can be in the prototype or definition, not both
 - best practice – in the prototype
 - must appear at the tail end of the parameter list
- Can have multiple default values
 - must appear consecutively at the tail end of the parameter list

Default Argument Values

Example - no default arguments

```
double calc_cost(double base_cost, double tax_rate);  
  
double calc_cost(double base_cost, double tax_rate) {  
    return base_cost += (base_cost * tax_rate);  
}  
  
int main() {  
    double cost {0};  
    cost = calc_cost(100.0, 0.06);  
    return 0;  
}
```

Default Argument Values

Example - single default argument

```
double calc_cost(double base_cost, double tax_rate = 0.06);

double calc_cost(double base_cost, double tax_rate) {
    return base_cost += (base_cost * tax_rate);
}

int main() {
    double cost {0};
    cost = calc_cost(200.0);    // will use the default tax
    cost = calc_cost(100.0, 0.08);      // will use 0.08 not the defauly
    return 0;
}
```

Default Argument Values

Example - multiple default arguments

```
double calc_cost(double base_cost, double tax_rate = 0.06, double shipping = 3.50);

double calc_cost(double base_cost, double tax_rate, double shipping) {
    return base_cost += (base_cost * tax_rate) + shipping;
}

int main() {
    double cost {0};
    cost = calc_cost(100.0, 0.08, 4.25); // will use no defaults
    cost = calc_cost(100.0, 0.08);      // will use default shipping
    cost = calc_cost(200.0);          // will use default tax and shipping
    return 0;
}
```

Overloading Functions

- We can have functions that have different parameter lists that have the same name
- Abstraction mechanism since we can just think 'print' for example
- A type of polymorphism
 - We can have the same name work with different data types to execute similar behavior
- The compiler must be able to tell the functions apart based on the parameter lists and argument supplied

Overloading Functions

Example

```
int add_numbers(int, int);      // add ints
double add_numbers(double, double); // add doubles

int main() {
    cout << add_numbers(10,20) << endl;      // integer
    cout << add_numbers(10.0, 20.0) << endl; // double
    return 0;
}
```

Overloading Functions

Example

```
int add_numbers(int a, int b) {  
    return a + b;  
}  
  
double add_numbers(double a, double b) {  
    return a + b;  
}
```

Overloading Functions

Example

```
void display(int n);  
void display(double d);  
void display(std::string s);  
void display(std::string s, std::string t);  
void display(std::vector<int> v);  
void display(std::vector<std::string> v);
```

Overloading Functions

Return type is not considered

```
int      get_value();  
double  get_value();  
  
// Error  
  
cout << get_value() << endl; // which one?
```

Passing Arrays To Functions

- We can pass an array to a function by providing square brackets in the formal parameter description

```
void print_array(int numbers []);
```

- The array elements are NOT copied
- Since the array name evaluates to the location of the array in memory - this address is what is copied
- So the function has no idea how many elements are in the array since all it knows is the location of the first element (the name of the array)

Example

```
void print_array(int numbers []);  
  
int main() {  
    int my_numbers[] {1,2,3,4,5};  
    print_array(my_numbers);  
    return 0;  
}  
  
void print_array(int numbers []) {  
    // Doesn't know how many elements are in the array???  
    // we need to pass in the size!!  
}
```

Example

```
void print_array(int numbers [], size_t size);

int main() {
    int my_numbers[] {1, 2, 3, 4, 5};
    print_array(my_numbers, 5);      / 1 2 3 4 5
    return 0;
}

void print_array(int numbers [], size_t size) {
    for (size_t i{0}; i < size; ++i)
        cout << numbers[i] << endl;
}
```

Example

- Since we are passing the location of the array
 - The function can modify the actual array!

```
void zero_array(int numbers [], size_t size) {  
    for (size_t i{0}; i < size; ++i )  
        numbers[i] = 0;                                // zero out array element  
}  
  
int main() {  
    int my_numbers[] {1,2,3,4,5};  
    zero_array(my_numbers, 5);                      // my_numbers is now zeroes!  
    print_array(my_numbers, 5);                        // 0 0 0 0 0  
    return 0;  
}
```

const parameters

- We can tell the compiler that function parameters are const (read-only)
- This could be useful in the print_array function since it should NOT modify the array

```
void print_array(const int numbers [], size_t size) {  
    for (size_t i{0}; i < size; ++i )  
        cout << numbers[i] << endl;  
numbers[i] = 0; // any attempt to modify the array  
                    // will result in a compiler error  
}
```

Pass by Reference

- Sometimes we want to be able to change the actual parameter from within the function body
- In order to achieve this we need the location or address of the actual parameter
- We saw how this is the effect with array, but what about other variable types?
- We can use reference parameters to tell the compiler to pass in a reference to the actual parameter.
- The formal parameter will now be an alias for the actual parameter

Example

```
void scale_number(int &num);           // prototype

int main() {
    int number {1000};
    scale_number(number);               // call
    cout << number << endl;           // 100
    return 0;
}

void scale_number(int &num) {           // definition
    if (num > 100)
        num = 100;
}
```

Example

```
void swap(int &a, int &b);  
  
int main() {  
    int x{10}, y{20};  
    cout << x << " " << y << endl; // 10 20  
    swap(x, y);  
    cout << x << " " << y << endl; // 20 10  
    return 0;  
}  
  
void swap(int &a, int &b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

vector example - pass by value

```
void print(std::vector<int> v);

int main() {
    std::vector<int> data {1,2,3,4,5};
    print(data);                                // 1 2 3 4 5
    return 0;
}

void print(std::vector<int> v) {
    for (auto num: v)
        cout << num << endl;
}
```

vector example - pass by reference

```
void print(std::vector<int> &v);  
  
int main() {  
    std::vector<int> data {1,2,3,4,5};  
    print(data); // 1 2 3 4 5  
    return 0;  
}  
  
void print(std::vector<int> &v) {  
    for (auto num: v)  
        cout << num << endl;  
}
```

vector example - pass by const reference

```
void print(const std::vector<int> &v) ;  
  
int main() {  
    std::vector<int> data {1,2,3,4,5};  
    print(data); // 1 2 3 4 5  
    return 0;  
}
```

```
void print(const std::vector<int> &v) {  
    v.at(0) = 200; // ERROR  
    for (auto num: v)  
        cout << num << endl;  
}
```

Scope Rules

- C++ uses scope rules to determine where an identifier can be used
- C++ uses static or lexical scoping
- Local or Block scope
- Global scope

Local or Block scope

- Identifiers declared in a block {}
- Function parameters have block scope
- Only visible within the block {} where declared
- Function local variables are only active while the function is executing
- Local variables are NOT preserved between function calls
- With nested blocks inner blocks can 'see' but outer blocks cannot 'see' in

Static local variables

- Declared with static qualifier

```
static int value {10};
```

- Value IS preserved between function calls
- Only initialized the first time the function is called

Global scope

- Identifier declared outside any function or class
- Visible to all parts of the program after the global identifier has been declared
- Global constants are OK
- Best practice – don't use global variables

How do Function Calls Work?

- **Functions use the 'function call stack'**
 - Analogous to a stack of books
 - LIFO – Last In First Out
 - push and pop
- **Stack Frame or Activation Record**
 - Functions must return control to function that called it
 - Each time a function is called we create a new activation record and push it on stack
 - When a function terminates we pop the activation record and return
 - Local variables and function parameters are allocated on the stack
- **Stack size is finite – Stack Overflow**

Inline Functions

- Function calls have a certain amount of overhead
- You saw what happens on the call stack
- Sometimes we have simple functions
- We can **suggest** to the compiler to compile them 'inline'
 - avoid function call overhead
 - generate inline assembly code
 - faster
 - could cause code bloat
- Compilers optimizations are very sophisticated
 - will likely inline even without your suggestion

Example

```
inline int add_numbers(int a, int b) { // definition
    return a + b;
}
```

```
int main() {
    int result {0};
    result = add_numbers(100,200); // call
    return 0;
}
```

Recursive Functions

- A recursive function is a function that calls itself
 - Either directly or indirectly through another function
- Recursive problem solving
 - Base case
 - Divide the rest of problem into subproblem and do recursive call
- There are many problems that lend themselves to recursive solutions
- Mathematic – factorial, Fibonacci, fractals,...
- Searching and sorting – binary search, search trees, ...

Example - Factorial

$$0! = 1$$

$$n! = n * (n-1)!$$

- **Base case:**
 - $\text{factorial}(0) = 1$
- **Recursive case:**
 - $\text{factorial}(n) = n * \text{factorial}(n-1)$

Example - Factorial

```
unsigned long long factorial(unsigned long long n) {  
    if (n == 0)  
        return 1;                                // base case  
    return n * factorial(n-1);                  // recursive case  
}  
  
int main() {  
    cout << factorial(8) << endl; // 40320  
    return 0;  
}
```

Example - Fibonacci

```
Fib(0) = 0
```

```
Fib(1) = 1
```

```
Fib(n) = Fib(n-1) + Fib(n-2)
```

- **Base case:**
 - $\text{Fib}(0) = 0$
 - $\text{Fib}(1) = 1$
- **Recursive case:**
 - $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

Example - Factorial

```
unsigned long long fibonacci(unsigned long long n) {  
    if (n <= 1)  
        return n;                                // base cases  
    return fibonacci(n-1) + fibonacci(n-2); // recursion  
}  
  
int main() {  
    cout << fibonacci(30) << endl; // 832040  
    return 0;  
}
```

Important notes

- If recursion doesn't eventually stop you will have infinite recursion
- Recursion can be resource intensive
- Remember the base case(s)
 - It terminates the recursion
- Only use recursive solutions when it makes sense
- Anything that can be done recursively can be done iteratively
 - Stack overflow error

Overview

- What is a pointer?
- Declaring pointers
- Storing addresses in pointers
- Dereferencing pointers
- Dynamic memory allocation
- Pointer arithmetic
- Pointers and arrays
- Pass-by-reference with pointers
- const and pointers
- Using pointers to functions
- Potential pointer pitfalls
- What is a reference?
- Review passing references to functions
- const and references
- Reference variables in range-based for loops
- Potential reference pitfalls
- Raw vs. Smart pointers

What is a Pointer?

- A variable
 - whose value is an address
- What can be at that address?
 - Another variable
 - A function
- Pointers point to variables or functions?
- If x is an integer variable and its value is 10 then I can declare a pointer that points to it
- To use the data that the pointer is pointing to you must know its type

Why use Pointers?

Can't I just use the variable or function itself?

Yes, but not always

- Inside functions, pointers can be used to access data that are defined outside the function.
Those variables may not be in scope so you can't access them by their name
- Pointers can be used to operate on arrays very efficiently
- We can allocate memory dynamically on the heap or free store.
 - This memory doesn't even have a variable name.
 - The only way to get to it is via a pointer
- With OO. pointers are how polymorphism works!
- Can access specific addresses in memory
 - useful in embedded and systems applications

Declaring Pointers

```
variable_type *pointer_name;
```

```
int *int_ptr;  
double* double_ptr;  
char *char_ptr;  
string *string_ptr;
```

Declaring Pointers

Initializing pointer variables to ‘point nowhere’

```
variable_type *pointer_name {nullptr};
```

```
int *int_ptr {};  
double* double_ptr {nullptr};  
char *char_ptr {nullptr};  
string *string_ptr {nullptr};
```

Declaring Pointers

Initializing pointer variables to ‘point nowhere’

- Always initialize pointers
- Uninitialized pointers contain garbage data and can ‘point anywhere’
- Initializing to zero or `nullptr` (C++ 11) represents address zero
 - implies that the pointer is ‘pointing nowhere’
- If you don’t initialize a pointer to point to a variable or function then you should initialize it to `nullptr` to ‘make it null’

Accessing Pointer Address?

& the address operator

- Variables are stored in unique addresses
- Unary operator
- Evaluates to the address of its operand
 - Operand cannot be a constant or expression that evaluates to temp values

```
int num{10};  
  
cout << "Value of num is: " << num << endl; // 10  
  
cout << "sizeof of num is: " << sizeof num << endl; // 4  
  
cout << "Address of num is: " << &num << endl; // 0x61ff1c
```

Accessing Pointer Address?

& the address operator - example

```
int *p;
```

```
cout << "Value of p is: " << p << endl; // 0x61ff60 - garbage
```

```
cout << "Address of p is: " << &p << endl; // 0x61ff18
```

```
cout << "sizeof of p is: " << sizeof p << endl; // 4
```

```
p = nullptr; // set p to point nowhere
```

```
cout << "Value of p is: " << p << endl; // 0
```

Accessing Pointer Addresses?

`sizeof` a pointer variable

- Don't confuse the size of a pointer and the size of what it points to
- All pointers in a program have the same size
- They may be pointing to very large or very small types

```
int *p1 {nullptr};  
double *p2 {nullptr};  
unsigned long long *p3 {nullptr};  
vector<string> *p4 {nullptr};  
string *p5 {nullptr};
```

Storing an Address in Pointer Variable?

Typed pointers

- The compiler will make sure that the address stored in a pointer variable is of the correct type

```
int score {10};  
double high_temp {100.7};  
  
int *score_ptr {nullptr};  
  
score_ptr = &score; // OK  
  
score_ptr = &high_temp; // Compiler Error
```

Storing an Address in Pointer Variable?

& the address operator

- Pointers are variables so they can change
- Pointers can be null
- Pointers can be uninitialized

```
double high_temp {100.7};  
double low_temp {37.2};
```

```
double *temp_ptr;
```

```
temp_ptr = &high_temp; // points to high_temp  
temp_ptr = &low_temp; // now points to low_temp
```

```
temp_ptr = nullptr;
```

Dereferencing a Pointer

Access the data we're pointing to – dereferencing a pointer

- If `score_ptr` is a pointer and has a valid address
- Then you can access the data at the address contained in the `score_ptr` using the dereferencing operator `*`

```
int score {100};  
int *score_ptr {&score};  
  
cout << *score_ptr << endl;      // 100  
  
*score_ptr = 200;  
cout << *score_ptr << endl;      // 200  
cout << score << endl;          // 200
```

Dereferencing a Pointer

Access the data we're pointing to

```
double high_temp {100.7};  
double low_temp {37.4};  
double *temp_ptr {&high_temp};
```

```
cout << *temp_ptr << endl; // 100.7
```

```
temp_ptr = &low_temp;
```

```
cout << *temp_ptr << endl; // 37.4
```

Dereferencing a Pointer

Access the data we're pointing to

```
string name { "Frank" };
```

```
string *string_ptr { &name };
```

```
cout << *string_ptr << endl; // Frank
```

```
name = "James";
```

```
cout << *string_ptr << endl; // James
```

Dynamic Memory Allocation

Allocating storage from the heap at runtime

- We often don't know how much storage we need until we need it
- We can allocate storage for a variable at run time
- Recall C++ arrays
 - We had to explicitly provide the size and it was fixed
 - But vectors grow and shrink dynamically
- We can use pointers to access newly allocated heap storage

Dynamic Memory Allocation

using **new** to allocate storage

```
int *int_ptr {nullptr};  
  
int_ptr = new int;           // allocate an integer on the heap  
  
cout << int_ptr << endl;    // 0x2747f28  
  
cout << *int_ptr << endl;   // 41188048 - garbage  
  
*int_ptr = 100;  
  
cout << *int_ptr << endl;   // 100
```

Dynamic Memory Allocation

using **delete** to deallocate storage

```
int *int_ptr {nullptr};
```

```
int_ptr = new int; // allocate an integer on the heap
```

```
• • •
```

```
delete int_ptr; // frees the allocated storage
```

Dynamic Memory Allocation

using `new[]` to allocate storage for an array

```
int *array_ptr {nullptr};  
int size {};  
  
cout << "How big do you want the array? ";  
cin >> size;  
  
array_ptr = new int[size]; // allocate array on the heap  
  
// We can access the array here  
// we'll see how in a few slides
```

Dynamic Memory Allocation

using `delete []` to deallocate storage for an array

```
int *array_ptr {nullptr};  
int size {};  
  
cout << "How big do you want the array?";  
cin >> size;  
  
array_ptr = new int[size]; // allocate array on the heap  
  
.  
.  
.  
  
delete [] array_ptr; // free allocated storage
```

Relationship Between Arrays and Pointers

- The value of an array name is the address of the first element in the array
- The value of a pointer variable is an address
- If the pointer points to the same data type as the array element then the pointer and array name can be used interchangeably (almost)

Relationship Between Arrays and Pointers

```
int scores[] {100, 95, 89};

cout << scores << endl;      // 0x61fec8
cout << *scores << endl;    // 100

int *score_ptr {scores};

cout << score_ptr << endl;  // 0x61fec8
cout << *score_ptr << endl; // 100
```

Relationship Between Arrays and Pointers

```
int scores[] {100, 95, 89};

int *score_ptr {scores};

cout << score_ptr[0] << endl; // 100

cout << score_ptr[1] << endl; // 95

cout << score_ptr[2] << endl; // 89
```

Using pointers in expressions

```
int scores[] {100, 95, 89};
```

```
int *score_ptr {scores};
```

```
cout << score_ptr << endl; // 0x61ff10
```

```
cout << (score_ptr + 1) << endl; // 0x61ff14
```

```
cout << (score_ptr + 2) << endl; // 0x61ff18
```

Using pointers in expressions

```
int scores[] {100, 95, 89};

int *score_ptr {scores};

cout << *score_ptr << endl; // 100

cout << *(score_ptr + 1) << endl; // 95

cout << *(score_ptr + 2) << endl; // 89
```

Subscript and Offset notation equivalence

```
int array_name[] {1, 2, 3, 4, 5};  
  
int *pointer_name {array_name};
```

Subscript Notation

array_name[index]

pointer_name[index]

Offset Notation

* (array_name + index)

* (pointer_name + index)

Pointer Arithmetic

- Pointers can be used in
 - Assignment expressions
 - Arithmetic expressions
 - Comparison expressions
- C++ allows pointer arithmetic
- Pointer arithmetic only makes sense with raw arrays

++ and --

- `(++)` increments a pointer to point to the next array element

`int_ptr++;`

- `(--)` decrements a pointer to point to the previous array element

`int_ptr--;`

+ and -

- (+) increment pointer by $n * \text{sizeof}(\text{type})$

`int_ptr += n; or int_ptr = int_ptr + n;`

- (-) decrement pointer by $n * \text{sizeof}(\text{type})$

`int_ptr -= n; or int_ptr = int_ptr - n;`

Subtracting two pointers

- Determine the number of elements between the pointers
- Both pointers must point to the same data type

```
int n = int_ptr2 - int_ptr1;
```

Comparing two pointers == and !=

Determine if two pointers point to the same location

- does NOT compare the data where they point!

```
string s1 {"Frank"};
string s2 {"Frank"};

string *p1 {&s1};
string *p2 {&s2};
string *p3 {&s1};

cout << (p1 == p2) << endl;      // false
cout << (p1 == p3) << endl;      // true
```

Comparing the data pointers point to

Determine if two pointers point to the same data

- you must compare the referenced pointers

```
string s1 {"Frank"};
string s2 {"Frank"};

string *p1 {&s1};
string *p2 {&s2};
string *p3 {&s1};

cout << (*p1 == *p2) << endl;      // true
cout << (*p1 == *p3) << endl;      // true
```

Passing pointers to a function

const and Pointers

- There are several ways to qualify pointers using const
 - Pointers to constants
 - Constant pointers
 - Constant pointers to constants

Pointers to constants

- The data pointed to by the pointers is constant and **cannot** be changed.
- The pointer itself can change and point somewhere else.

```
int high_score {100};  
int low_score { 65};  
const int *score_ptr { &high_score };  
  
*score_ptr = 86;      // ERROR  
score_ptr = &low_score; // OK
```

Constant pointers

- The data pointed to by the pointers can be changed.
- The pointer itself **cannot** change and point somewhere else

```
int high_score {100};  
int low_score { 65};  
int *const score_ptr { &high_score };  
  
*score_ptr = 86;      // OK  
score_ptr = &low_score; // ERROR
```

Constant pointers to constants

- The data pointed to by the pointer is constant and **cannot** be changed.
- The pointer itself **cannot** change and point somewhere else.

```
int high_score {100};  
int low_score { 65};  
const int *const score_ptr { &high_score };  
  
*score_ptr = 86;      // ERROR  
score_ptr = &low_score; // ERROR
```

Passing pointers to a function

- Pass-by-reference with pointer parameters
- We can use pointers and the dereference operator to achieve pass-by-reference
- The function parameter is a pointer
- The actual parameter can be a pointer or address of a variable

Passing pointers to a function

Pass-by-reference with pointers – defining the function

```
void double_data(int *int_ptr);  
  
void double_data(int *int_ptr) {  
    *int_ptr *= 2;  
  
    // *int_ptr = *int_ptr * 2;  
}
```

Passing pointers to a function

Pass-by-reference with pointers – calling the function

```
int main() {  
    int value {10};  
  
    cout << value << endl;      // 10  
  
    double_data( &value );  
  
    cout << value << endl;      // 20  
}
```

Returning a Pointer from a Function

- Functions can also return pointers

```
type *function();
```

- Should return pointers to
 - Memory dynamically allocated in the function
 - To data that was passed in
- Never return a pointer to a local function variable!

returning a parameter

```
int *largest_int(int *int_ptr1, int *int_ptr2) {  
    if (*int_ptr1 > *int_ptr2)  
        return int_ptr1;  
    else  
        return int_ptr2;  
}
```

returning a parameter

```
int main() {  
    int a{100};  
    int b{200};  
  
    int *largest_ptr {nullptr};  
largest_ptr = largest_int(&a, &b);  
    cout << *largest_ptr << endl;      // 200  
    return 0;  
}
```

returning dynamically allocated memory

```
int *create_array(size_t size, int init_value = 0) {  
  
    int *new_storage {nullptr};  
  
    new_storage = new int[size];  
    for (size_t i{0}; i < size; ++i)  
        * (new_storage + i) = init_value;  
    return new_storage;  
}
```

returning dynamically allocated memory

```
int main() {  
    int *my_array;          // will be allocated by the function  
  
    my_array = create_array(100,20); // create the array  
  
    // use it  
  
    delete [] my_array; /   / be sure to free the storage  
    return 0;  
}
```

Never return a pointer to a local variable!!

```
int *dont_do_this () {  
    int size {};  
    . . .  
    return &size;  
}  
  
int *or_this () {  
    int size {};  
    int *int_ptr {&size};  
    . . .  
    return int_ptr;  
}
```

Potential Pointer Pitfalls

- Uninitialized pointers
- Dangling Pointers
- Not checking if new failed to allocate memory
- Leaking memory

Uninitialized pointers

```
int *int_ptr; // pointing anywhere  
.  
.  
.  
  
*int_ptr = 100; // Hopefully a crash
```

Dangling pointer

- Pointer that is pointing to released memory
 - For example, 2 pointers point to the same data
 - 1 pointer releases the data with delete
 - The other pointer accesses the release data
- Pointer that points to memory that is invalid
 - We saw this when we returned a pointer to a function local variable

Not checking if new failed

- If `new` fails an exception is thrown
- We can use exception handling to catch exceptions
- Dereferencing a null pointer will cause your program to crash

Leaking memory

- Forgetting to release allocated memory with `delete`
- If you lose your pointer to the storage allocated on the heap you have no way to get to that storage again
- The memory is orphaned or leaked
- One of the most common pointer problems

What is a Reference?

- An alias for a variable
- Must be initialized to a variable when declared
- Cannot be null
- Once initialized cannot be made to refer to a different variable
- Very useful as function parameters
- Might be helpful to think of a reference as a constant pointer that is automatically dereferenced

What is a reference?

Using references in range-based for loop

```
vector<string> stooges {"Larry", "Moe", "Curly"};
```

```
for (auto str: stooges)
    str = "Funny";      // changes the copy
```

```
for (auto str:stooges)
    cout << str << endl;    // Larry, Moe, Curly
```

What is a reference?

Using references in range-based for loop

```
vector<string> stooges {"Larry", "Moe", "Curly"};
```

```
for (auto &str: stooges)
    str = "Funny"; // changes the actual
```

```
for (auto str:stooges)
    cout << str << endl; // Funny, Funny, Funny
```

What is a reference?

Using references in range-based for loop

```
vector<string> stooges {"Larry", "Moe", "Curly"};
```

```
for (auto const &str: stooges)
    str = "Funny"; // compiler error
```

What is a reference?

Using references in range-based for loop

```
vector<string> stooges {"Larry", "Moe", "Curly";  
  
for (auto const &str:stooges)  
    cout << str << endl; // Larry, Moe, Curly
```

What is a reference?

Passing references to functions

- Please refer to the section 11 videos and examples

l-values

- l-values
 - values that have names and are addressable
 - modifiable if they are not constants

```
int x {100};      // x is an l-value  
x = 1000;  
x = 1000 + 20;
```

```
string name;      // name is an l-value  
name = "Frank";
```

I-values

- I-values
 - values that have names and are addressable
 - modifiable if they are not constants

```
100 = x;           // 100 is NOT an l-value
(1000 + 20) = x; // (1000 + 20) is NOT an l-value
```

```
string name;
name = "Frank";
"Frank" = name; // "Frank" is NOT an l-value
```

r-values

- r-value (non-addressable and non-assignable)
 - A value that's not an l-value
 - on the right-hand side of an assignment expression
 - a literal
 - a temporary which is intended to be non-modifiable

```
int x {100};                                // 100 is an r-value
int y = x + 200;                            // (x+200) is an r-value
```

```
string name;
name = "Frank";                            // "Frank" is an r-value
```

```
int max_num = max(20,30); // max(20,30) is an r-value
```

r-values

- r-values can be assigned to l-values explicitly

```
int x {100};
```

```
int y {0};
```

```
y = 100;           // r-value 100 assigned to l-value y
```

```
x = x + y;     // r-value (x+y) assigned to l-value x
```

I-value references

- The references we've used are I-value references
 - Because we are referencing I-values

```
int x {100};
```

```
int &ref1 = x;      // ref1 is reference to l-value
ref1 = 1000;
```

```
int &ref2 = 100; // Error 100 is an r-value
```

I-value references

- The same when we pass-by-reference

```
int square(int &n) {  
    return n*n;  
}
```

```
int num {10};
```

```
square(num); // OK
```

```
square(5); // Error - can't reference r-value 5
```

When to use pointers vs. references parameters

- Pass-by-value
 - when the function does **not** modify the actual parameter, and
 - the parameter is small and efficient to copy like simple types (int, char, double, etc.)

When to use pointers vs. references parameters

- Pass-by-reference using a pointer
 - when the function does modify the actual parameter, and
 - the parameter is expensive to copy, and
 - Its OK to the pointer is allowed a nullptr value

When to use pointers vs. references parameters

- Pass-by-reference using a pointer to const
 - when the function does **not** modify the actual parameter, and
 - the parameter is expensive to copy, and
 - Its OK to the pointer is allowed a nullptr value

When to use pointers vs. references parameters

- Pass-by-reference using a const pointer to const
 - when the function does **not** modify the actual parameter, and
 - the parameter is expensive to copy, and
 - Its OK to the pointer is allowed a nullptr value, and
 - You don't want to modify the pointer itself

When to use pointers vs. references parameters

- Pass-by-reference using a reference
 - when the function **does** modify the actual parameter,
and
 - the parameter is expensive to copy,
and
 - The parameter will never be nullptr

When to use pointers vs. references parameters

- Pass-by-reference using a const reference
 - when the function does **not** modify the actual parameter, and
 - the parameter is expensive to copy, and
 - The parameter will never be nullptr

Object-Oriented Programming – Classes and Objects

- What is Object-Oriented Programming?
- What are Classes and Objects?
- Declaring Classes and creating Objects
- Dot and pointer operators
- public and private access modifiers
- Methods, Constructors and Destructors
 - class methods
 - default and overloaded constructors
 - copy and move constructors
 - shallow vs. deep copying
 - this pointer
- static class members
- struct vs. class
- friend of a class

What is Object-Oriented Programming?

- Procedural Programming
- Procedural Programming limitations
- OO Programming concepts and their advantages
- OO Programming limitations

Procedural programming

- Focus is on processes or actions that a program takes
- Programs are typically a collection of functions
- Data is declared separately
- Data is passed as arguments into functions
- Fairly easy to learn

Procedural programming - Limitations

- Functions need to know the structure of the data.
 - if the structure of the data changes many functions must be changed
- As programs get larger they become more:
 - difficult to understand
 - difficult to maintain
 - difficult to extend
 - difficult to debug
 - difficult to reuse code
 - fragile and easier to break

What is Object-Oriented Programming?

- Classes and Objects
 - focus is on classes that model real-world domain entities
 - allows developers to think at a higher level of abstraction
 - used successfully in very large programs

What is Object-Oriented Programming?

- Encapsulation
 - objects contain data AND operations that work on that data
 - Abstract Data Type (ADT)

What is Object-Oriented Programming?

- Information-hiding
 - implementation-specific logic can be hidden
 - users of the class code to the interface since they don't need to know the implementation
 - more abstraction
 - easier to test, debug, maintain and extend

What is Object-Oriented Programming?

- Reusability
 - easier to reuse classes in other applications
 - faster development
 - higher quality

What is Object-Oriented Programming?

- Inheritance
 - can create new classes in term of existing classes
 - reusability
 - polymorphic classes
- Polymorphism and more...

Limitations

- Not a panacea
 - OO Programming won't make bad code better
 - not suitable for all types of problems
 - not everything decomposes to a class
- Learning curve
 - usually a steeper learning curve, especially for C++
 - many OO languages, many variations of OO concepts
- Design
 - usually more up-front design is necessary to create good models and hierarchies
- Programs can be:
 - larger in size
 - slower
 - more complex

Classes and Objects

- **Classes**
 - blueprint from which objects are created
 - a user-defined data-type
 - has attributes (data)
 - has methods (functions)
 - can hide data and methods
 - provides a public interface
- **Example classes**
 - Account
 - Employee
 - Image
 - std::vector
 - std::string

Classes and Objects

- **Objects**
 - created from a class
 - represents a specific instance of a class
 - can create many, many objects
 - each has its own identity
 - each can use the defined class methods
- **Example Account objects**
 - Frank's account is an instance of an Account
 - Jim's account is an instance of an Account
 - Each has its own balance, can make deposits, withdrawals, etc.

Classes and Objects

```
int high_score;  
int low_score;
```

```
Account frank_account;
```

```
Account jim_account;
```

```
std::vector<int> scores;  
std::string name;
```

Declaring a Class

```
class Class_Name  
{  
    // declaration(s);  
};
```

Player

```
class Player
{
    // attributes
    std::string name;
    int health;
    int xp;

    // methods
    void talk(std::string text_to_say);
    bool is_dead();
};
```

Creating objects

```
Player frank;
```

```
Player hero;
```

```
Player *enemy = new Player();  
delete enemy;
```

Account

```
class Account
{
    // attributes
    std::string name;
    double balance;

    // methods
    bool withdraw(double amount);
    bool deposit(double amount);

};
```

Creating objects

```
Account frank_account;
```

```
Account jim_account;
```

```
Account *mary_account = new Account();  
delete mary_account;
```

Creating objects

```
Account frank_account;
```

```
Account jim_account;
```

```
Account accounts[] {frank_account, jim_account};
```

```
std::vector<Account> accounts1 {frank_account};  
accounts1.push_back(jim_account);
```

Accessing Class Members

- We can access
 - class attributes
 - class methods
- Some class members will not be accessible (more on that later)
- We need an object to access instance variables

Accessing Class Members

If we have an object (dot operator)

- **Using the dot operator**

```
Account frank_account;
```

```
frank_account.balance;  
frank_account.deposit(1000.00);
```

Accessing Class Members

If we have a pointer to an object (member of pointer operator)

- Dereference the pointer then use the dot operator.

```
Account *frank_account = new Account();
```

```
(*frank_account).balance;  
(*frank_account).deposit(1000.00);
```

- Or use the member of pointer operator (arrow operator)

```
Account *frank_account = new Account();
```

```
frank_account->balance;  
frank_account->deposit(1000.00);
```

Class Member Access Modifiers

public, private, and protected

- public
 - accessible everywhere
- private
 - accessible only by members or friends of the class
- protected
 - used with inheritance – we'll talk about it in the next section

Class Member Access Modifiers

public

```
class Class_Name  
{  
public:  
    // declaration(s);  
};
```

Class Member Access Modifiers

private

```
class Class_Name
{
private:
    // declaration(s);

}
```

Class Member Access Modifiers

protected

```
class Class_Name
{
protected:
    // declaration(s);

}
```

Declaring a Class

Player

```
class Player
{
    private:
        std::string name;
        int health;
        int xp;
    public:
        void talk(std::string text_to_say);
        bool is_dead();
};
```

Creating objects

```
Player frank;  
frank.name = "Frank";           // Compiler error  
frank.health = 1000;            // Compiler error  
frank.talk("Ready to battle"); // OK
```

```
Player *enemy = new Player();  
enemy->xp = 100;                // Compiler error  
enemy->talk("I will hunt you down"); // OK
```

```
delete enemy;
```

Declaring a Class

Account

```
class Account
{
private:
    std::string name;
    double balance;

public:
    bool withdraw(double amount);
    bool deposit(double amount);
};
```

Creating objects

```
Account frank_account;  
frank_account.balance = 10000000.00; // Compiler error  
frank_account.deposit(10000000.0); // OK  
frank_account.name = "Frank's Account"; // Compiler error  
  
Account *mary_account = new Account();  
  
mary_account->balance = 10000.0; // Compiler error  
mary_account->withdraw(10000.0); // OK  
  
delete mary_account;
```

Implementing Member Methods

- Very similar to how we implemented functions
- Member methods have access to member attributes
 - So you don't need to pass them as arguments!
- Can be implemented inside the class declaration
 - Implicitly inline
- Can be implemented outside the class declaration
 - Need to use `Class_name::method_name`
- Can separate specification from implementation
 - .h file for the class declaration
 - .cpp file for the class implementation

Implementing Member Methods

Inside the class declaration

```
class Account {  
  
private:  
    double balance;  
public:  
    void set_balance(double bal) {  
        balance = bal;  
    }  
    double get_balance() {  
        return balance;  
    }  
};
```

Implementing Member Methods

Outside the class declaration

```
class Account {  
  
private:  
    double balance;  
public:  
    void set_balance(double bal);  
    double get_balance();  
};  
  
void Account::set_balance(double bal) {  
    balance = bal;  
}  
double Account::get_balance() {  
    return balance;  
}
```

Separating Specification from Implementation

Account.h

```
class Account {  
  
private:  
    double balance;  
public:  
    void set_balance(double bal);  
    double get_balance();  
};
```

Separating Specification from Implementation

Include Guards

```
#ifndef _ACCOUNT_H_
#define _ACCOUNT_H_
```

```
// Account class declaration
```

```
#endif
```

Separating Specification from Implementation

Account.h

```
#ifndef _ACCOUNT_H_
#define _ACCOUNT_H_

class Account {

private:
    double balance;
public:
    void set_balance(double bal);
    double get_balance();
};

#endif
```

Separating Specification from Implementation

Account.h - #pragma once

#pragma once

```
class Account {  
  
private:  
    double balance;  
public:  
    void set_balance(double bal);  
    double get_balance();  
};
```

Separating Specification from Implementation

Account.cpp

```
#include "Account.h"

void Account::set_balance(double bal) {
    balance = bal;
}

double Account::get_balance() {
    return balance;
}
```

Separating Specification from Implementation

main.cpp

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

int main() {
    Account frank_account;
    frank_account.set_balance(1000.00);
    double bal = frank_account.get_balance();

    std::cout << bal << std::endl; // 1000
    return 0;
}
```

Constructors

- Special member method
- Invoked during object creation
- Useful for initialization
- Same name as the class
- No return type is specified
- Can be overloaded

Player Constructors

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
    // Overloaded Constructors
    Player();
    Player(std::string name);
    Player(std::string name, int health, int xp);
};
```

Account Constructors

```
class Account
{
private:
    std::string name;
    double balance;
public:
    // Constructors
    Account();
    Account(std::string name, double balance);
    Account(std::string name);
    Account(double balance);
};
```

Destructors

- Special member method
- Same name as the class proceeded with a tilde (~)
- Invoked automatically when an object is destroyed
- No return type and no parameters
- Only 1 destructor is allowed per class – cannot be overloaded!
- Useful to release memory and other resources

Player Destructor

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
    Player();
    Player(std::string name);
    Player(std::string name, int health, int xp);
// Destructor
~Player();
};
```

Account Destructor

```
class Account
{
private:
    std::string name;
    double balance;
public:
    Account();
    Account(std::string name, double balance);
    Account(std::string name);
    Account(double balance);
    // Destructor
    ~Account();
};
```

Creating objects

```
{  
    Player slayer;  
    Player frank {"Frank", 100, 4 } ;  
    Player hero {"Hero"} ;  
    Player villain {"Villain"} ;  
    // use the objects  
} // 4 destructors called  
  
Player *enemy = new Player("Enemy", 1000, 0) ;  
delete enemy; // destructor called
```

The Default Constructor

- Does not expect any arguments
 - Also called the no-args constructor
- If you write no constructors at all for a class
 - C++ will generate a Default Constructor that does nothing
- Called when you instantiate a new object with no arguments

```
Player frank;  
Player *enemy = new Player;
```

Declaring a Class

Account - using default constructor

```
class Account
{
private:
    std::string name;
    double balance;
public:
    bool withdraw(double amount);
    bool deposit(double amount);
};
```

Creating objects

Using the default constructor

```
Account frank_account;
```

```
Account jim_account;
```

```
Account *mary_account = new Account;  
delete mary_account;
```

Declaring a Class

Account - user-defined no-args constructor

```
class Account
{
private:
    std::string name;
    double balance;
public:
    Account() {
        name = "None";
        balance = 0.0;
    }
    bool withdraw(double amount);
    bool deposit(double amount);
};
```

Declaring a Class

Account - no default constructor

```
class Account
{
private:
    std::string name;
    double balance;
public:
    Account(std::string name_val, double bal) {
        name = name_val;
        balance = bal;
    }
    bool withdraw(double amount);
    bool deposit(double amount);
};
```

Creating objects

Using the default constructor

```
Account frank_account; // Error  
Account jim_account; // Error  
  
Account *mary_account = new Account; // Error  
delete mary_account;  
  
Account bill_account {"Bill", 15000.0}; // OK
```

Overloading Constructors

- Classes can have as many constructors as necessary
- Each must have a unique signature
- Default constructor is no longer compiler-generated once another constructor is declared

Constructors and Destructors

Overloaded Constructors

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
    // Overloaded Constructors
    Player();
    Player(std::string name_val);
    Player(std::string name_val, int health_val, int xp_val);
};
```

Constructors and Destructors

Overloaded Constructors

```
Player::Player() {  
    name = "None";  
    health = 0;  
    xp = 0;  
}
```

```
Player::Player(std::string name_val) {  
    name = name_val;  
    health = 0;  
    xp = 0;  
}
```

Constructors and Destructors

Overloaded Constructors

```
Player::Player(std::string name_val, int health_val, int
xp_val) {
    name = name_val;
    health = health_val;
    xp = xp_val;
}
```

Creating objects

```
Player empty;                                // None, 0, 0

Player hero {"Hero"};                         // Hero, 0, 0
Player villain {"Villain"};                   // Villain, 0, 0

Player frank {"Frank", 100, 4};               // Frank, 100, 4

Player *enemy = new Player("Enemy", 1000, 0); // Enemy, 1000, 0
delete enemy;
```

Constructor Initialization Lists

- So far, all data member values have been set in the constructor body
- Constructor initialization lists
 - are more efficient
 - initialization list immediately follows the parameter list
 - initializes the data members as the object is created!
 - order of initialization is the order of declaration in the class

Constructor Initialization Lists

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
    // Overloaded Constructors
    Player();
    Player(std::string name_val);
    Player(std::string name_val, int health_val, int xp_val);
};
```

Constructor Initialization Lists

Player()

Previous way:

```
Player::Player() {  
    name = "None";      // assignment not initialization  
    health = 0;  
    xp = 0;  
}
```

Better way:

```
Player::Player()  
: name{"None"}, health{0}, xp{0} {  
}
```

Constructor Initialization Lists

Player(std::string)

Previous way:

```
Player::Player(std::string name_val) {  
    name = name_val;    // assignment not initialization  
    health = 0;  
    xp = 0;  
}
```

Better way:

```
Player::Player(std::string name_val)  
: name{name_val}, health{0}, xp{0} {  
}
```

Constructor Initialization Lists

Player(std::string, int, int)

Previous way:

```
Player::Player(std::string name_val, int health_val, int xp_val) {  
    name = name_val; // assignment not initialization  
    health = health_val;  
    xp = xp_val;  
}
```

Better way:

```
Player::Player(std::string name_val, int health_val, int xp_val)  
: name{name_val}, health{health_val}, xp{xp_val} {  
}
```

Constructor Initialization Lists

```
Player::Player()  
: name{"None"}, health{0}, xp{0} {  
}
```

```
Player::Player(std::string name_val)  
: name{name_val}, health{0}, xp{0} {  
}
```

```
Player::Player(std::string name_val, int health_val, int  
xp_val)  
: name{name_val}, health{health_val}, xp{xp_val} {  
}
```

Delegating Constructors

- Often the code for constructors is very similar
- Duplicated code can lead to errors
- C++ allows delegating constructors
 - code for one constructor can call another in the initialization list
 - avoids duplicating code

Delegating Constructors

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
    // Overloaded Constructors
    Player();
    Player(std::string name_val);
    Player(std::string name_val, int health_val, int xp_val);
};
```

Delegating Constructors

```
Player::Player()  
: name{"None"}, health{0}, xp{0} {  
}
```

```
Player::Player(std::string name_val)  
: name{name_val}, health{0}, xp{0} {  
}
```

```
Player::Player(std::string name_val, int health_val, int  
xp_val)  
: name{name_val}, health{health_val}, xp{xp_val} {  
}
```

Delegating Constructors

```
Player::Player(std::string name_val, int health_val, int xp_val)
    : name{name_val}, health{health_val}, xp{xp_val} {
```

```
}
```

```
Player::Player(std::string name_val)
    : Player {"None", 0, 0} {
```

```
}
```

Default Constructor Parameters

- Can often simplify our code and reduce the number of overloaded constructors
- Same rules apply as we learned with non-member functions

Default Constructor Parameters

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
    // Constructor with default parameter values
    Player(std::string name_val = "None",
           int health_val = 0,
           int xp_val = 0);
};
```

Default Constructor Parameters

```
Player::Player(std::string name_val, int health_val, int
xp_val)
    : name {name_val}, health {health_val}, xp {xp_val} {

}

Player empty;                                // None, 0, 0
Player frank {"Frank"};                      // Frank, 0, 0
Player villain {"Villain", 100, 55}; // Villain, 100, 55
Player hero {"Hero", 100};                  // Hero, 100, 0

// Note what happens if you declare a no-args constructor
```

Copy Constructor

- When objects are copied C++ must create a new object from an existing object
- When is a copy of an object made?
 - passing object by value as a parameter
 - returning an object from a function by value
 - constructing one object based on another of the same class
- C++ must have a way of accomplishing this so it provides a compiler-defined copy constructor if you don't

Pass object by-value

```
Player hero { "Hero", 100, 20};
```

```
void display_player(Player p) {
    // p is a COPY of hero in this example
    // use p
    // Destructor for p will be called
}
```

```
display_player(hero);
```

Return object by value

```
Player enemy;

Player create_super_enemy() {
    Player an_enemy{"Super Enemy", 1000, 1000};
    return an_enemy; // A COPY of an_enemy is returned
}

enemy = create_super_enemy();
```

Construct one object based on another

```
Player hero { "Hero", 100, 100};
```

```
Player another_hero {hero}; // A COPY of hero is made
```

Copy Constructor

- If you don't provide your own way of copying objects by value then the compiler provides a default way of copying objects
- Copies the values of each data member to the new object
 - default memberwise copy
- Perfectly fine in many cases
- Beware if you have a pointer data member
 - Pointer will be copied
 - Not what it is pointing to
 - Shallow vs. Deep copy – more in the next video

Best practices

- Provide a copy constructor when your class has raw pointer members
- Provide the copy constructor with a **const reference** parameter
- Use STL classes as they already provide copy constructors
- Avoid using raw pointer data members if possible

Declaring the Copy Constructor

```
Type::Type(const Type &source);
```

```
Player::Player(const Player &source);
```

```
Account::Account(const Account &source);
```

Implementing the Copy Constructor

```
Type::Type(const Type &source) {  
    // code or initialization list to copy the object  
}
```

Implementing the Copy Constructor

Player

```
Player::Player(const Player &source)
: name{source.name},
  health {source.health},
  xp {source.xp} {
}
```

Implementing the Copy Constructor

Account

```
Account::Account(const Account &source)
: name{source.name},
  balance {source.balance} {
```

}

Shallow vs. Deep Copying

- Consider a class that contains a pointer as a data member
- Constructor allocates storage dynamically and initializes the pointer
- Destructor releases the memory allocated by the constructor
- What happens in the default copy constructor?

Default copy constructor

- memberwise copy
- Each data member is copied from the source object
- The pointer is copied NOT what it points to (shallow copy)
- **Problem:** when we release the storage in the destructor, the other object still refers to the released storage!

Copy Constructor

Shallow

```
class Shallow {  
private:  
    int *data;                                // POINTER  
public:  
    Shallow(int d);                           // Constructor  
    Shallow(const Shallow &source);           // Copy  
Constructor  
    ~Shallow();                               // Destructor  
};
```

Copy Constructor

Shallow constructor

```
Shallow::Shallow(int d) {  
    data = new int; // allocate storage  
    *data = d;  
}
```

Copy Constructor

Shallow destructor

```
Shallow::~Shallow() {  
    delete data; // free storage  
    cout << "Destructor freeing data" << endl;  
}
```

Copy Constructor

Shallow copy constructor

```
Shallow::Shallow(const Shallow &source)
: data(source.data) {
    cout << "Copy constructor - shallow"
    << endl;
}
```

Shallow copy - only the pointer is copied - not what it is pointing to!

Problem: source and the newly created object BOTH point to the SAME data area!

Copy Constructor

Shallow - a simple method that expects a copy

```
Shallow::Shallow(const Shallow &source)
: data(source.data) {
    cout << "Copy constructor - shallow"
    << endl;
}
```

Shallow copy - only the pointer is copied - not what it is pointing to!

Problem: source and the newly created object BOTH point to the SAME data area!

Copy Constructor

Sample main - will likely crash

```
int main() {  
    Shallow obj1 {100};  
    display_shallow(obj1);  
    // obj1's data has been released!  
  
    obj1.set_data_value(1000);  
    Shallow obj2 {obj1};  
    cout << "Hello world" << endl;  
    return 0;  
}
```

User-provided copy constructor

- Create a **copy** of the pointed-to data
- Each copy will have a pointer to unique storage in the heap
- Deep copy when you have a raw pointer as a class data member

Copy Constructor

Deep

```
class Deep {  
private:  
    int *data;           // POINTER  
public:  
    Deep(int d);        // Constructor  
    Deep(const Deep &source); // Copy Constructor  
    ~Deep();            // Destructor  
};
```

Copy Constructor

Deep constructor

```
Deep::Deep(int d) {  
    data = new int; // allocate storage  
    *data = d;  
}
```

Copy Constructor

Deep constructor

```
Deep::~Deep() {  
    delete data; // free storage  
    cout << "Destructor freeing data" << endl;  
}
```

Copy Constructor

Deep constructor

```
Deep::Deep(const Deep &source)
{
    data = new int;    // allocate storage
    *data = *source.data;
    cout << "Copy constructor - deep"
        << endl;
}
```

Deep copy - create new storage and copy values

Copy Constructor

Deep copy constructor – delegating constructor

```
Deep::Deep(const Deep &source)
: Deep(*source.data) {
    cout << "Copy constructor - deep"
    << endl;
}
```

Delegate to Deep (int) and pass in the int (*source.data) source is pointing to

Copy Constructor

Deep – a simple method that expects a copy

```
void display_deep(Deep s) {  
    cout << s.get_data_value() << endl;  
}
```

When *s* goes out of scope the destructor is called and releases data.

No Problem: since the storage being released is unique to *s*

Copy Constructor

Sample main – will not crash

```
int main() {  
    Deep obj1 {100};  
    display_deep(obj1);  
  
    obj1.set_data_value(1000);  
    Deep obj2 {obj1};  
  
    return 0;  
}
```

Move Constructor

- Sometimes when we execute code the compiler creates unnamed temporary values

```
int total { 0 };  
total = 100 + 200;
```

- $100 + 200$ is evaluated and 300 stored in an unnamed temp value
 - the 300 is then stored in the variable total
 - then the temp value is discarded
-
- The same happens with objects as well

When is it useful?

- Sometimes copy constructors are called many times automatically due to the copy semantics of C++
- Copy constructors doing deep copying can have a significant performance bottleneck
- C++11 introduced move semantics and the move constructor
- Move constructor moves an object rather than copy it
- Optional but recommended when you have a raw pointer
- Copy elision – C++ may optimize copying away completely (RVO-Return Value Optimization)

r-value references

- Used in moving semantics and perfect forwarding
- Move semantics is all about r-value references
- Used by move constructor and move assignment operator to efficiently move an object rather than copy it
- R-value reference operator (&&)

r-value references

```
int x {100}
int &l_ref = x;      // l-value reference
l_ref = 10;          // change x to 10
```

```
int &&r_ref = 200;    // r-value ref
r_ref = 300;          // change r_ref to 300
```

```
int &&x_ref = x;    // Compiler error
```

l-value reference parameters

```
int x {100};           // x is an l-value
```

```
void func(int &num); // A
```

```
func(x);      // calls A - x is an l-value
```

```
func(200);      // Error - 200 is an r-value
```

error: cannot bind non-const lvalue reference of type 'int&' to an rvalue of type 'int'

r-value reference parameters

```
int x {100};           // x is an l-value
```

```
void func(int &&num); // B
```

```
func(200);      // calls B - 200 is an r-value
```

```
func(x);       // ERROR - x is an l-value
```

error: cannot bind rvalue reference of type 'int&&' to lvalue of type 'int'

l-value and r-value reference parameters

```
int x {100};           // x is an l-value

void func(int &num);   // A
void func(int &&num); // B

func(x);      // calls A - x is an l-value
func(200);    // calls B - 200 is an r-value
```

Example - Move class

```
class Move {  
private:  
    int *data;           // raw pointer  
public:  
    void set_data_value(int d) { *data = d; }  
    int get_data_value()      { return *data; }  
    Move(int d);          // Constructor  
    Move(const Move &source); // Copy Constructor  
    ~Move();              // Destructor  
};
```

Move class copy constructor

```
Move::Move(const Move &source) {  
    data = new int;  
    *data = *source.data;  
}
```

Allocate storage and copy

Inefficient copying

```
Vector<Move> vec;  
  
vec.push_back(Move{10});  
vec.push_back(Move{20});
```

Copy Constructors will be called to copy the temps

Inefficient copying

Constructor for: 10

Constructor for: 10

Copy constructor - deep copy for: 10

Destructor freeing data for: 10

Constructor for: 20

Constructor for: 20

Copy constructor - deep copy for: 20

Constructor for: 10

Copy constructor - deep copy for: 10

Destructor freeing data for: 10

Destructor freeing data for: 20

What does it do?

- Instead of making a deep copy of the move constructor
 - ‘moves’ the resource
 - Simply copies the address of the resource from source to the current object
 - And, nulls out the pointer in the source pointer
- Very efficient

syntax - r-value reference

```
Type::Type(Type &&source);
```

```
Player::Player(Player &&source);
```

```
Move::Move(Move &&source);
```

Move class with move constructor

```
class Move {  
private:  
    int *data;           // raw pointer  
public:  
    void set_data_value(int d) { *data = d; }  
    int get_data_value() { return *data; }  
    Move(int d);        // Constructor  
    Move(const Move &source); // Copy Constructor  
Move(Move &&source); // Move Constructor  
    ~Move();            // Destructor  
};
```

Move class move constructor

```
Move::Move (Move &&source)
: data{source.data} {
    source.data = nullptr;
}
```

'Steal' the data and then null out the source pointer

efficient

```
Vector<Move> vec;  
  
vec.push_back(Move{10});  
vec.push_back(Move{20});
```

Move Constructors will be called for the temp r-values

efficient

Constructor for: 10

Move constructor - moving resource: 10

Destructor freeing data for nullptr

Constructor for: 20

Move constructor - moving resource: 20

Move constructor - moving resource: 10

Destructor freeing data for nullptr

Destructor freeing data for nullptr

Destructor freeing data for: 10

Destructor freeing data for: 20

this pointer

- this is a reserved keyword
- Contains the address of the object - so it's a pointer to the object
- Can only be used in class scope
- All member access is done via the this pointer
- Can be used by the programmer
 - To access data member and methods
 - To determine if two objects are the same (more in the next section)
 - Can be dereferenced (*this) to yield the current object

this pointer

```
void Account::set_balance(double bal) {  
    balance = bal; // this->balance is implied  
}
```

this pointer

To disambiguate identifier use

```
void Account::set_balance(double balance) {  
    balance = balance; // which balance? The parameter  
}
```

```
void Account::set_balance(double balance) {  
    this->balance = balance; // Unambiguous  
}
```

this pointer

To determine object identity

- Sometimes it's useful to know if two objects are the same object

```
int Account::compare_balance(const Account &other) {  
    if (this == &other)  
        std::cout << "The same objects" << std::endl;  
    ...  
}  
  
frank_account.compare_balance(frank_account);
```

- We'll use the this pointer again when we overload the assignment operator

Using const with classes

- Pass arguments to class member methods as const
- We can also create const objects
- What happens if we call member functions on const objects?
- const-correctness

Creating a const object

- villain is a const object so its attributes cannot change

```
const Player villain {"Villain", 100, 55};
```

Using const with classes

What happens when we call member methods on a const object?

```
const Player villain {"Villain", 100, 55};  
  
villain.set_name("Nice guy"); // ERROR  
  
std::cout << villain.get_name() << std::endl; // ERROR
```

Using const with classes

What happens when we call member methods on a const object?

```
const Player villain {"Villain", 100, 55};
```

```
void display_player_name(const Player &p) {  
    cout << p.get_name() << endl;  
}
```

```
display_player_name(villain);           // ERROR
```

Using const with classes

const methods

```
class Player {  
private:  
    . . .  
public:  
    std::string get_name() const;  
    . . .  
};
```

Using const with classes

const-correctness

```
const Player villain {"Villain", 100, 55};  
  
villain.set_name("Nice guy"); // ERROR  
  
std::cout << villain.get_name() << std::endl; // OK
```

Using const with classes

const methods

```
class Player {  
private:  
    . . .  
public:  
    std::string get_name() const;  
    // ERROR if code in get_name modifies this object  
    . . .  
};
```

Static Class Members

- Class data members can be declared as static
 - A single data member that belongs to the class, not the objects
 - Useful to store class-wide information
- Class functions can be declared as static
 - Independent of any objects
 - Can be called using the class name

Player class -static members

```
class Player {  
private:  
    static int num_players;  
  
public:  
    static int get_num_players();  
    . . .  
};
```

Player class – initialize the static data

Typically in Player.cpp

```
#include "Player.h"

int Player::num_players = 0;
```

Player class – implement static method

```
int Player::get_num_players() {  
    return num_players;  
}
```

Player class -update the constructor

```
Player::Player(std::string name_val, int health_val,  
int xp_val)  
: name{name_val}, health{health_val}, xp{xp_val} {  
    ++num_players;  
}
```

Player class - Destructor

```
Player::~Player() {  
    --num_players;  
}
```

main.cpp

```
void display_active_players() {  
    cout << "Active players: "  
        << Player::get_num_players() << endl;  
}  
  
int main() {  
    display_active_players();  
  
    Player obj1 {"Frank"};  
    display_active_players();  
    . . .
```

Structs vs Classes

- In addition to define a class we can declare a struct
- struct comes from the C programming language
- Essentially the same as a class except
 - members are public by default

class

```
class Person {  
    std::string name;  
    std::string get_name();  
};
```

```
Person p;  
p.name = "Frank";    // compiler error - private  
std::cout << p.get_name(); // compiler error - private
```

struct

```
struct Person {  
    std::string name;  
    std::string get_name(); // Why if name is public?  
};
```

```
Person p;  
p.name = "Frank"; // OK - public  
std::cout << p.get_name(); // OK - public
```

Some general guidelines

- struct
 - Use struct for passive objects with public access
 - Don't declare methods in struct

- class
 - Use class for active objects with private access
 - Implement getters/setters as needed
 - Implement member methods as needed

Friends of a Class

- Friend

- A function or class that has access to private class member
- And, that function or class is NOT a member of the class it is accessing

- Function

- Can be regular non-member functions
- Can be member methods of another class

- Class

- Another class can have access to private class members

Friends of a Class

- Friendship must be granted NOT taken
 - Declared explicitly in the class that is granting friendship
 - Declared in the function prototype with the keyword `friend`
- Friendship is not symmetric
 - Must be explicitly granted
 - if A is a friend of B
 - B is NOT a friend of A
- Friendship is not transitive
 - Must be explicitly granted
 - if A is a friend of B AND
 - B is a friend of C
 - then A is NOT a friend of C

non-member function

```
class Player {  
    friend void display_player(Player &p);  
    std::string name;  
    int health;  
    int xp;  
public:  
    . . .  
};
```

non-member function

```
void display_player(Player &p) {  
    std::cout << p.name << std::endl;  
    std::cout << p.health << std::endl;  
    std::cout << p.xp << std::endl;  
}
```

display_player may also change private data members

member function of another class

```
class Player {  
    friend void Other_class::display_player(Player &p);  
    std::string name;  
    int health;  
    int xp;  
public:  
    . . .  
};
```

member function of another class

```
class Other_class {  
    . . .  
public:  
    void display_player(Player &p) {  
        std::cout << p.name << std::endl;  
        std::cout << p.health << std::endl;  
        std::cout << p.xp << std::endl;  
    }  
};
```

Another class as a friend

```
class Player {  
    friend class Other_class;  
    std::string name;  
    int health;  
    int xp;  
public:  
    . . .  
};
```

Section Overview

Operator Overloading

- What is Operator Overloading?
- Overloading the assignment operator (=)
 - Copy semantics
 - Move semantics
- Overloading operators as member functions
- Overloading operators as global functions
- Overloading stream insertion (<<) and extraction operators (>>)

Operator Overloading

What is Operator Overloading?

- Using traditional operators such as +, =, *, etc. with user-defined types
- Allows user defined types to behave similar to built-in types
- Can make code more readable and writable
- Not done automatically (except for the assignment operator)
They must be explicitly defined

Operator Overloading

What is Operator Overloading?

Suppose we have a Number class that models any number

- Using functions:

```
Number result = multiply(add(a,b),divide(c,d));
```

- Using member methods:

```
Number result = (a.add(b)).multiply(c.divide(d));
```

Operator Overloading

What is Operator Overloading?

Suppose we have a Number class that models any number

- Using overloaded operators

```
Number result = (a+b) * (c/d) ;
```

Operator Overloading

What operators can be overloaded?

- The majority of C++'s operators can be overloaded
- The following operators cannot be overloaded

operator
::
: ?
. *
.
sizeof

Operator Overloading

Some basic rules

- Precedence and Associativity cannot be changed
- 'arity' cannot be changed (i.e. can't make the division operator unary)
- Can't overload operators for primitive type (e.g. int, double, etc.)
- Can't create new operators
- [], (), ->, and the assignment operator (=) **must** be declared as member methods
- Other operators can be declared as member methods or global functions

Operator Overloading

Some examples

- **int**

```
a = b + c  
a < b  
std::cout << a
```

- **double**

```
a = b + c  
a < b  
std::cout << a
```

- **long**

```
a = b + c  
a < b  
std::cout << a
```

- **std::string**

```
s1 = s2 + s3  
S1 < s2  
std::cout << s1
```

- **Mystring**

```
s1 = s2 + s3  
s1 < s2  
s1 == s2  
std::cout << s1
```

- **Player**

```
p1 < p2  
p1 == p2  
std::cout << p1
```

Operator Overloading

Mystring class declaration

```
class Mystring {  
  
private:  
    char *str; // C-style string  
  
public:  
    Mystring();  
    Mystring(const char *s);  
    Mystring(const Mystring &source);  
    ~Mystring();  
    void display() const;  
    int get_length() const;  
    const char *get_str() const;  
};
```

Operator Overloading

Copy assignment operator (=)

- C++ provides a default assignment operator used for assigning one object to another

```
Mystring s1 {"Frank"};  
Mystring s2 = s1;      // NOT assignment  
                      // same as Mystring s2{s1};  
  
s2 = s1;              // assignment
```

- Default is memberwise assignment (shallow copy)
 - If we have raw pointer data member we must deep copy

Operator Overloading

Overloading the copy assignment operator (deep copy)

```
Type &Type::operator=(const Type &rhs);
```

```
Mystring &Mystring::operator=(const Mystring &rhs);
```

```
s2 = s1; // We write this
```

```
s2.operator=(s1); // operator= method is called
```

Operator Overloading

Overloading the copy assignment operator (deep copy)

```
Mystring &Mystring::operator=(const Mystring &rhs) {  
    if (this == &rhs)  
        return *this;  
  
    delete [] str;  
    str = new char[std::strlen(rhs.str) + 1];  
    std::strcpy(str, rhs.str);  
  
    return *this;  
}
```

Operator Overloading

Overloading the copy assignment operator – steps for deep copy

- Check for self assignment

```
if (this == &rhs)    // p1 = p1;  
    return *this;    // return current object
```

- Deallocate storage for this->str since we are overwriting it

```
delete [] str;
```

Operator Overloading

Overloading the copy assignment operator – steps for deep copy

- Allocate storage for the deep copy

```
str = new char [std::strlen(rhs.str) + 1];
```

Operator Overloading

Overloading the copy assignment operator – steps for deep copy

- Perform the copy

```
std::strcpy(str, rhs.str);
```

- Return the current by reference to allow chain assignment

```
return *this;
```

```
// s1 = s2 = s3;
```

Operator Overloading

Move assignment operator (=)

- You can choose to overload the move assignment operator
 - C++ will use the copy assignment operator if necessary

```
Mystring s1;
```

```
s1 = Mystring {"Frank"}; // move assignment
```

- If we have raw pointer we should overload the move assignment operator for efficiency

Operator Overloading

Overloading the Move assignment operator

```
Type &Type::operator=(Type &&rhs);
```

```
Mystring &Mystring::operator=(Mystring &&rhs);
```

```
s1 = Mystring{"Joe"}; // move operator= called
```

```
s1 = "Frank";           // move operator= called
```

Operator Overloading

Overloading the Move assignment operator

```
Mystring &Mystring::operator=(Mystring &&rhs) {  
  
    if (this == &rhs)          // self assignment  
        return *this;         // return current object  
  
    delete [] str;            // deallocate current storage  
    str = rhs.str;           // steal the pointer  
  
    rhs.str = nullptr;        // null out the rhs object  
  
    return *this;             // return current object  
}
```

Operator Overloading

Overloading the Move assignment operator – steps

- Check for self assignment

```
if (this == &rhs)
    return *this;           // return current object
```

- Deallocate storage for this->str since we are overwriting it

```
delete [] str;
```

Operator Overloading

Overloading the Move assignment operator – steps for deep copy

- Steal the pointer from the rhs object and assign it to this->str

```
str = rhs.str;
```

- Null out the rhs pointer

```
rhs.str = nullptr;
```

- Return the current object by reference to allow chain assignment

```
return *this;
```

Operator Overloading

Unary operators as member methods (++, --, -, !)

```
ReturnType Type::operatorOp();
```

```
Number Number::operator-() const;
Number Number::operator++();           // pre-increment
Number Number::operator++(int);        // post-increment
bool Number::operator!() const;

Number n1 {100};
Number n2 = -n1;                      // n1.operator-()
n2 = ++n1;                          // n1.operator++()
n2 = n1++;                           // n1.operator++(int)
```

Operator Overloading

Mystring operator- make lowercase

```
Mystring larry1 {"LARRY"};  
Mystring larry2;  
  
larry1.display();           // LARRY  
  
larry2 = -larry1;          // larry1.operator-()  
  
larry1.display();          // LARRY  
larry2.display();          // larry
```

Operator Overloading

Mystring operator- make lowercase

```
Mystring Mystring::operator-() const {
    char *buff = new char[std::strlen(str) + 1];
    std::strcpy(buff, str);
    for (size_t i=0; i<std::strlen(buff); i++)
        buff[i] = std::tolower(buff[i]);
    Mystring temp {buff};
    delete [] buff;
    return temp;
}
```

Operator Overloading

Binary operators as member methods (+, -, ==, !=, <, >, etc.)

```
ReturnType Type::operatorOp(const Type &rhs);
```

```
Number Number::operator+(const Number &rhs) const;
```

```
Number Number::operator-(const Number &rhs) const;
```

```
bool Number::operator==(const Number &rhs) const;
```

```
bool Number::operator<(const Number &rhs) const;
```

```
Number n1 {100}, n2 {200};
```

```
Number n3 = n1 + n2; // n1.operator+(n2)
```

```
n3 = n1 - n2; // n1.operator-(n2)
```

```
if (n1 == n2) . . . // n1.operator==(n2)
```

Operator Overloading

Mystring operator==

```
bool Mystring::operator==(const Mystring &rhs) const {
    if (std::strcmp(str, rhs.str) == 0)
        return true;
    else
        return false;
}

// if (s1 == s2)          // s1 and s2 are Mystring objects
```

Operator Overloading

Mystring operator+ (concatenation)

```
Mystring larry {"Larry"};  
Mystring moe {"Moe"};  
Mystring stooges {" is one of the three stooges"};  
  
Mystring result = larry + stooges;  
                    // larry.operator+(stooges);  
  
result = moe + " is also a stooge";  
                    // moe.operator+("is also a stooge");  
  
result = "Moe" + stooges; // "Moe".operator+(stooges) ERROR
```

Operator Overloading

Mystring operator+ (concatenation)

```
Mystring Mystring::operator+(const Mystring &rhs) const {
    size_t buff_size = std::strlen(str) +
                      std::strlen(rhs.str) + 1;
    char *buff = new char[buff_size];
    std::strcpy(buff, str);
    std::strcat(buff, rhs.str);
    Mystring temp {buff};
    delete [] buff;
    return temp;
}
```

Operator Overloading

Unary operators as global functions (++ , -- , - , !)

```
ReturnType operatorOp(Type &obj);
```

```
Number operator- (const Number &obj);  
Number operator++(Number &obj); // pre-increment  
Number operator++(Number &obj, int); // post-increment  
bool operator! (const Number &obj);
```

```
Number n1 {100};  
Number n2 = -n1; // operator- (n1)  
n2 = ++n1; // operator++ (n1)  
n2 = n1++; // operator++ (n1, int)
```

Operator Overloading

Mystring operator- make lowercase

```
Mystring larry1 {"LARRY"};  
Mystring larry2;  
  
larry1.display();                                // LARRY  
  
larry2 = -larry1;                                 // operator-(larry1)  
  
larry1.display();                                // LARRY  
larry2.display();                                // larry
```

Operator Overloading

Mystring operator-

- Often declared as **friend** functions in the class declaration

```
Mystring operator-(const Mystring &obj) {
    char *buff = new char[std::strlen(obj.str) + 1];
    std::strcpy(buff, obj.str);
    for (size_t i=0; i<std::strlen(buff); i++)
        buff[i] = std::tolower(buff[i]);
    Mystring temp {buff};
    delete [] buff;
    return temp;
}
```

Operator Overloading

Binary operators as global functions (+, -, ==, !=, <, >, etc.)

```
ReturnType operatorOp(const Type &lhs, const Type &rhs);
```

```
Number operator+(const Number &lhs, const Number &rhs);
```

```
Number operator-(const Number &lhs, const Number &rhs);
```

```
bool operator==(const Number &lhs, const Number &rhs);
```

```
bool operator<(const Number &lhs, const Number &rhs);
```

```
Number n1 {100}, n2 {200};
```

```
Number n3 = n1 + n2; // operator+(n1, n2)
```

```
n3 = n1 - n2; // operator-(n1, n2)
```

```
if (n1 == n2) . . . // operator==(n1, n2)
```

Operator Overloading

```
Mystring operator==
```

```
bool operator==(const Mystring &lhs, const Mystring &rhs) {
    if (std::strcmp(lhs.str, rhs.str) == 0)
        return true;
    else
        return false;
}
```

- If declared as a friend of Mystring can access private str attribute
- Otherwise we must use getter methods

Operator Overloading

Mystring operator+ (concatenation)

```
Mystring larry {"Larry"};  
Mystring moe {"Moe"};  
Mystring stooges {" is one of the three stooges"};  
  
Mystring result = larry + stooges;  
                    // operator+(larry, stooges);  
  
result = moe + " is also a stooge";  
                    // operator+(moe, "is also a stooge");  
  
result = "Moe" + stooges; // OK with non-member functions
```

Operator Overloading

Mystring operator+ (concatenation)

```
Mystring operator+(const Mystring &lhs, const Mystring &rhs) {  
    size_t buff_size = std::strlen(lhs.str) +  
                      std::strlen(rhs.str) + 1;  
    char *buff = new char[buff_size];  
    std::strcpy(buff, lhs.str);  
    std::strcat(buff, rhs.str);  
    Mystring temp {buff};  
    delete [] buff;  
    return temp;  
}
```

Operator Overloading

stream insertion and extraction operators (<<, >>)

```
Mystring larry {"Larry"};
```

```
cout << larry << endl; // Larry
```

```
Player hero {"Hero", 100, 33};
```

```
cout << hero << endl; // {name: Hero, health: 100, xp:33}
```

Operator Overloading

stream insertion and extraction operators (<<, >>)

```
Mystring larry;
```

```
cin >> larry;
```

```
Player hero;
```

```
cin >> hero;
```

Operator Overloading

stream insertion and extraction operators (<<, >>)

- Doesn't make sense to implement as member methods
 - Left operand must be a user-defined class
 - Not the way we normally use these operators

```
Mystring larry;  
larry << cout; // huh?
```

```
Player hero;  
hero >> cin; // huh?
```

Operator Overloading

stream insertion operator (<<)

```
std::ostream &operator<<(std::ostream &os, const Mystring &obj) {  
    os << obj.str;      // if friend function  
    // os << obj.get_str(); // if not friend function  
    return os;  
}
```

- Return a reference to the ostream so we can keep inserting
- Don't return ostream by value!

Operator Overloading

stream extraction operator (>>)

```
std::istream &operator>>(std::istream &is, Mystring &obj) {  
    char *buff = new char[1000];  
    is >> buff;  
    obj = Mystring{buff}; // If you have copy or move assignment  
    delete [] buff;  
    return is;  
}
```

- Return a reference to the istream so we can keep inserting
- Update the object passed in

Section Overview

Inheritance

- What is Inheritance?
 - Why is it useful?
- Terminology and Notation
- Inheritance vs. Composition
- Deriving classes from existing classes
 - Types of inheritance
- Protected members and class access
- Constructors and Destructors
 - Passing arguments to base class constructors
 - Order of constructor and destructors calls
- Redefining base class methods
- Class Hierarchies
- Multiple Inheritance

Inheritance

What is it and why is it used?

- Provides a method for creating new classes from existing classes
- The new class contains the data and behaviors of the existing class
- Allow for reuse of existing classes
- Allows us to focus on the common attributes among a set of classes
- Allows new classes to modify behaviors of existing classes to make it unique
 - Without actually modifying the original class

Inheritance

Related classes

- Player, Enemy, Level Boss, Hero, Super Player, etc.
- Account, Savings Account, Checking Account, Trust Account, etc.
- Shape, Line, Oval, Circle, Square, etc.
- Person, Employee, Student, Faculty, Staff, Administrator, etc.

Inheritance

Accounts

- Account
 - **balance, deposit, withdraw, ...**
- Savings Account
 - **balance, deposit, withdraw, interest rate, ...**
- Checking Account
 - **balance, deposit, withdraw, minimum balance, per check fee, ...**
- Trust Account
 - **balance, deposit, withdraw, interest rate, ...**

Inheritance

Accounts – without inheritance – code duplication

```
class Account {  
    // balance, deposit, withdraw, . . .  
};  
  
class Savings_Account {  
    // balance, deposit, withdraw, interest rate, . . .  
};  
  
class Checking_Account {  
    // balance, deposit, withdraw, minimum balance, per check fee, . . .  
};  
  
class Trust_Account {  
    // balance, deposit, withdraw, interest rate, . . .  
};
```

Inheritance

Accounts – with inheritance – code reuse

```
class Account {  
    // balance, deposit, withdraw, . . .  
};  
  
class Savings_Account : public Account {  
    // interest rate, specialized withdraw, . . .  
};  
  
class Checking_Account : public Account {  
    // minimum balance, per check fee, specialized withdraw, . . .  
};  
  
class Trust_Account : public Account {  
    // interest rate, specialized withdraw, . . .  
};
```

Inheritance

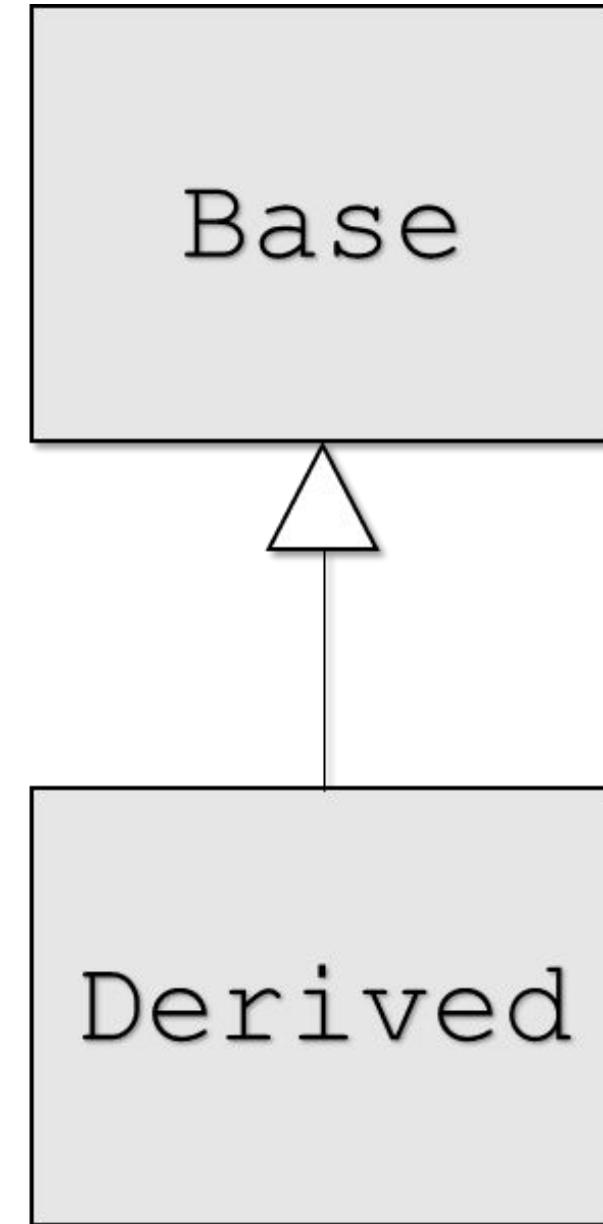
Terminology

- Inheritance
 - Process of creating new classes from existing classes
 - Reuse mechanism
- Single Inheritance
 - A new class is created from another ‘single’ class
- Multiple Inheritance
 - A new class is created from two (or more) other classes

Inheritance

Terminology

- Base class (parent class, super class)
 - The class being extended or inherited from
- Derived class (child class, sub class)
 - The class being created from the Base class
 - Will inherit attributes and operations from Base class



Inheritance

Terminology

- “Is-A” relationship
 - Public inheritance
 - Derived classes are sub-types of their Base classes
 - Can use a derived class object wherever we use a base class object
- Generalization
 - Combining similar classes into a single, more general class based on common attributes
- Specialization
 - Creating new classes from existing classes proving more specialized attributes or operations
- Inheritance or Class Hierarchies
 - Organization of our inheritance relationships

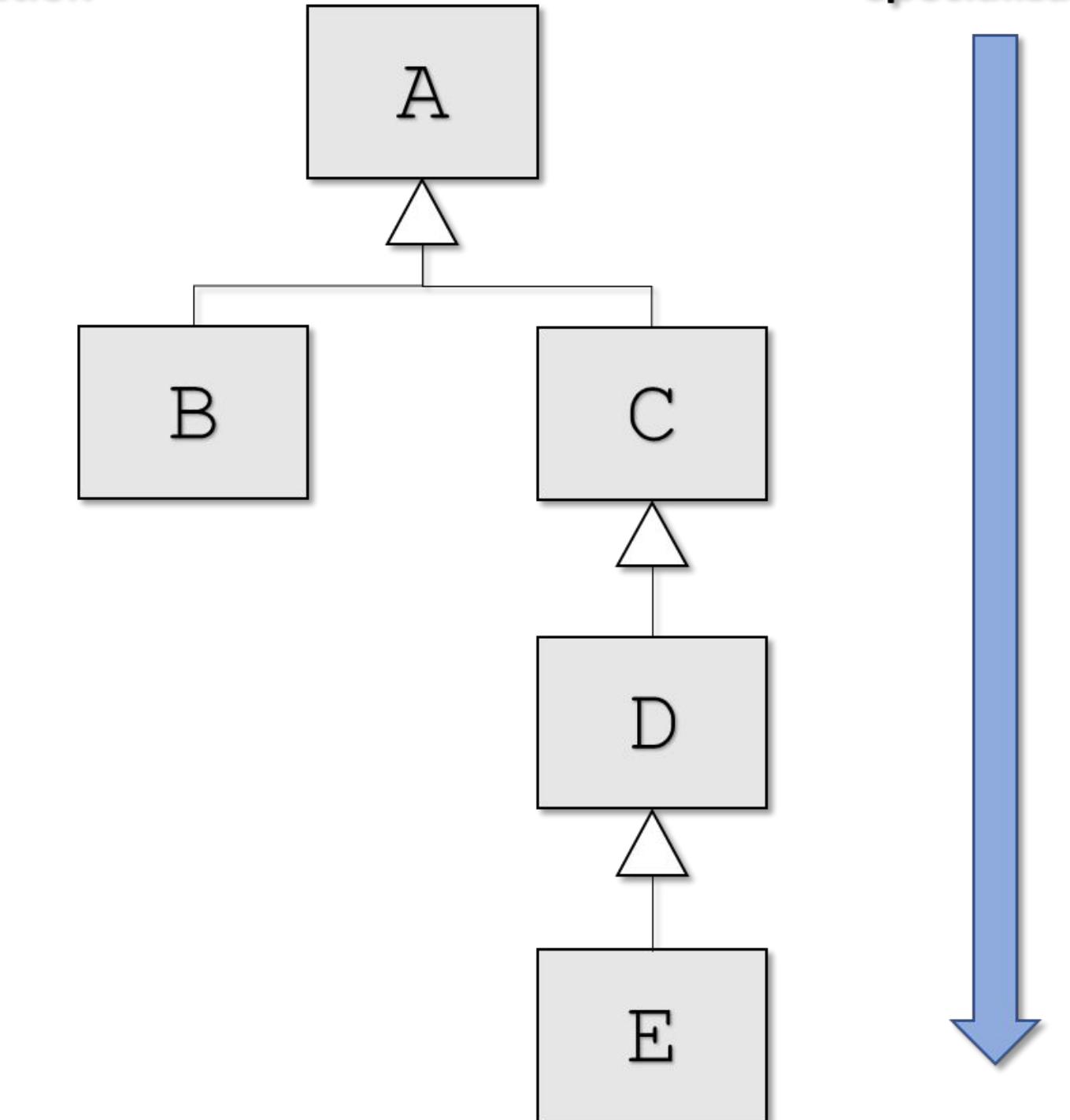
Inheritance

Class hierarchy

Classes:

- A
- B is derived from A
- C is derived from A
- D is derived from C
- E is derived from D

Generalization



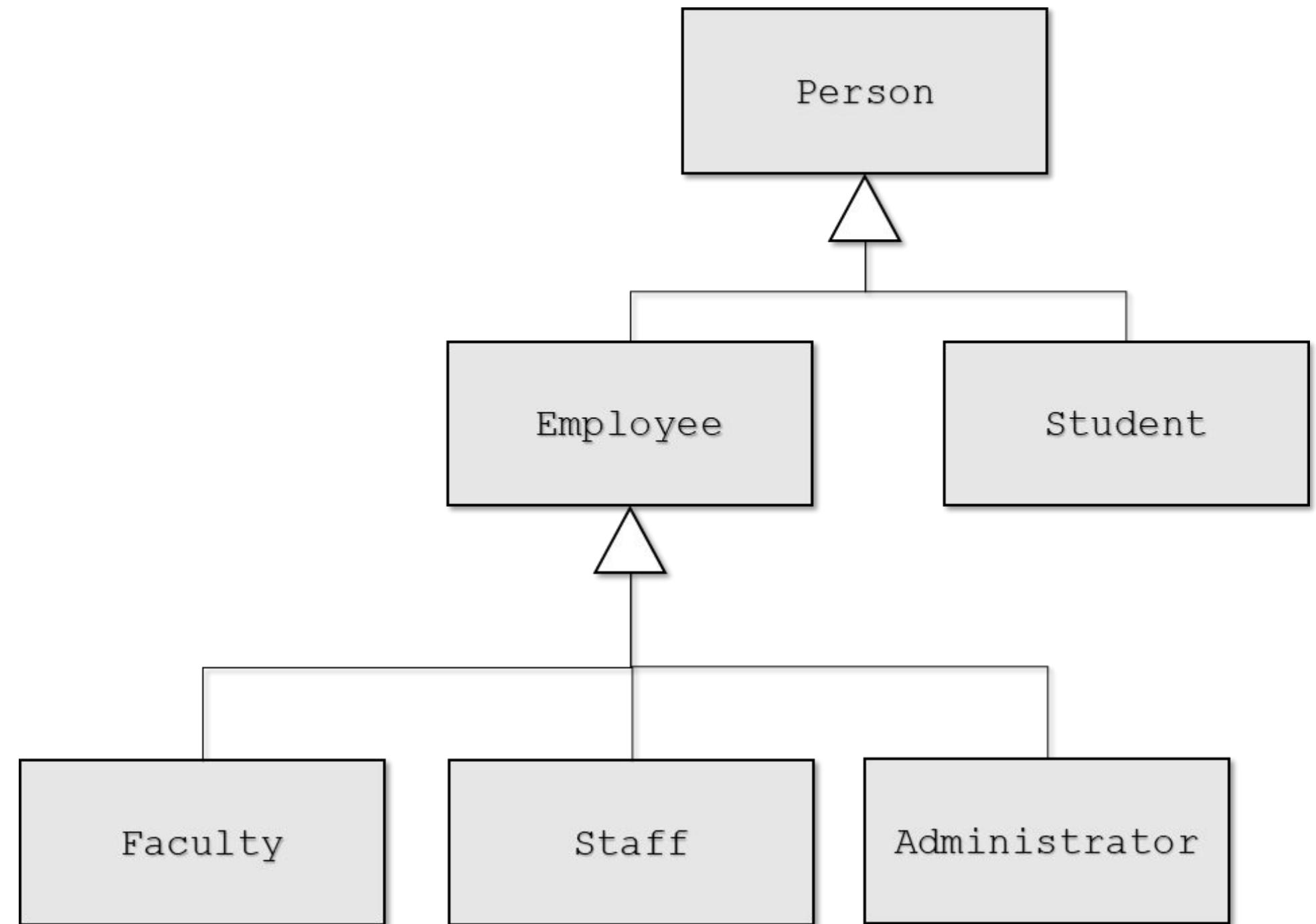
Specialization

Inheritance

Class hierarchy

Classes:

- Person
- Employee is derived from Person
- Student is derived from Person
- Faculty is derived from Employee
- Staff is derived from Employee
- Administrator is derived from Employee



Inheritance

Public Inheritance vs. Composition

- Both allow reuse of existing classes

- Public Inheritance

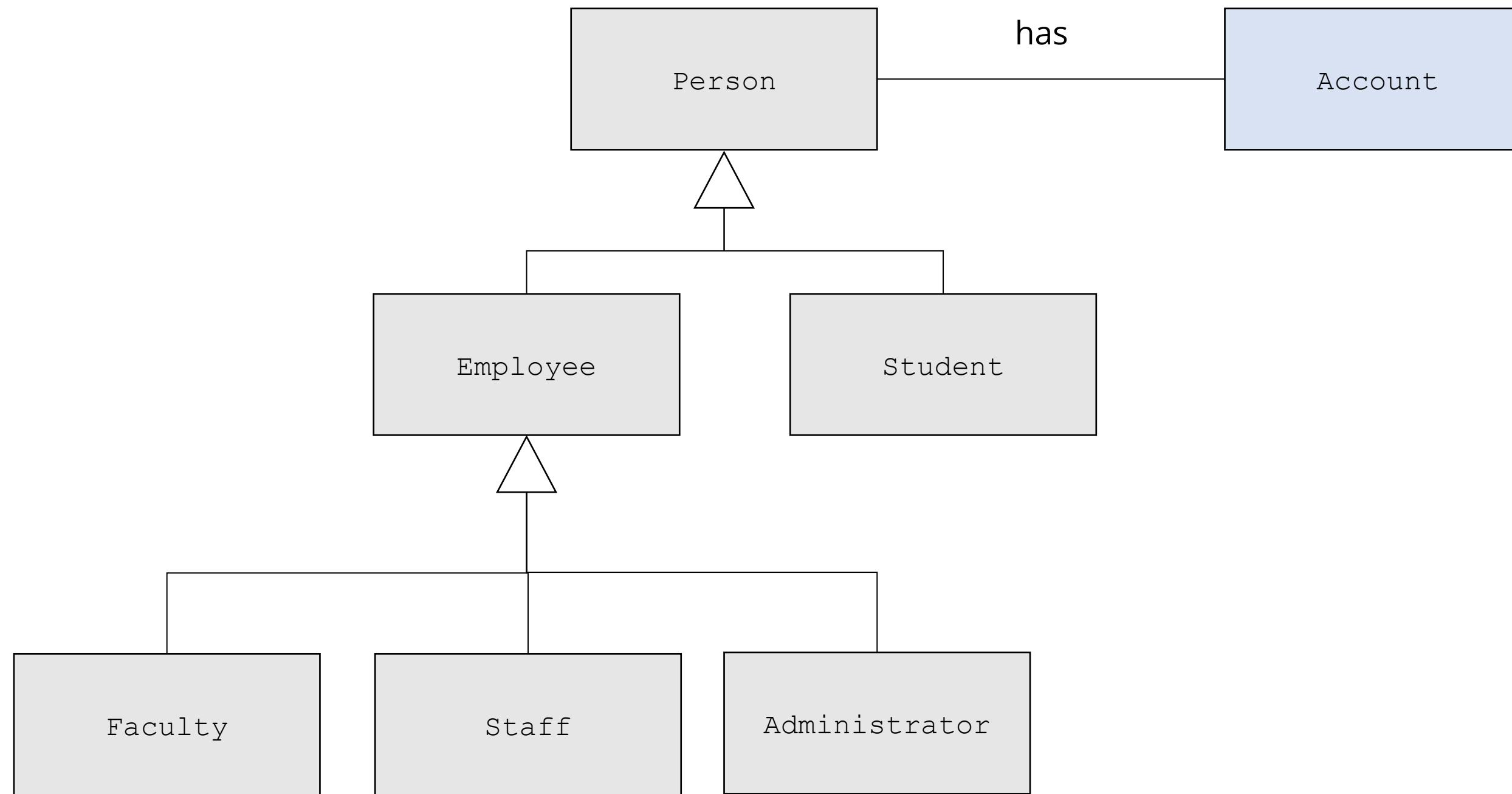
- “is-a” relationship
 - Employee ‘is-a’ Person
 - Checking Account ‘is-a’ Account
 - Circle “is-a” Shape

- Composition

- “has-a” relationship
 - Person “has a” Account
 - Player “has-a” Special Attack
 - Circle “has-a” Location

Inheritance

Public Inheritance vs. Composition



Inheritance

Public Inheritance vs. Composition

```
class Person {  
private:  
    std::string name; // has-a name  
    Account account; // has-a account  
};
```

Deriving classes from existing classes

C++ derivation syntax

```
class Base {  
    // Base class members . . .  
};
```

```
class Derived: access-specifier Base {  
    // Derived class members . . .  
};
```

Access-specifier can be: public, private, or protected

Deriving classes from existing classes

Types of inheritance in C++

- public
 - Most common
 - Establishes '**is-a**' relationship between Derived and Base classes
- private and protected
 - Establishes "derived class **has a** base class" relationship
 - "Is implemented in terms of" relationship
 - Different from composition

Deriving classes from existing classes

C++ derivation syntax

```
class Account {  
    // Account class members . . .  
};
```

```
class Savings_Account: public Account {  
    // Savings_Account class members . . .  
};
```

Savings_Account ‘is-a’ Account

Deriving classes from existing classes

C++ creating objects

```
Account account {};  
Account *p_account = new Account();  
  
account.deposit(1000.0);  
p_account->withdraw(200.0);  
  
delete p_account;
```

Deriving classes from exiting classes

C++ creating objects

```
Savings_Account sav_account {};  
Savings_Account *p_sav_account = new Savings_Account();  
  
sav_account.deposit(1000.0);  
p_sav_account->withdraw(200.0);  
  
delete p_sav_account;
```

Protected Members and Class Access

The protected class member modifier

```
class Base {  
  
    protected:  
        // protected Base class members . . .  
};
```

- Accessible from the Base class itself
- Accessible from classes Derived from Base
- Not accessible by objects of Base or Derived

Protected Members and Class Access

The protected class member modifier

```
class Base {  
public:  
    int a; // public Base class members . . .  
  
protected:  
    int b; // protected Base class members . . .  
  
private:  
    int c; // private Base class members . . .  
};
```

Deriving classes from existing classes

Access with **public** inheritance

Base Class

```
public: a  
protected: b  
private: c
```

public
inheritance

Access in
Derived Class

```
public: a  
protected: b  
c : no access
```

Deriving classes from existing classes

Access with **protected** inheritance

Base Class

```
public: a  
protected: b  
private: c
```

protected
inheritance

Access in Derived Class

```
protected: a  
protected: b  
c : no access
```

Deriving classes from existing classes

Access with **private** inheritance

Base Class

```
public: a  
protected: b  
private: c
```

private
inheritance

Access in
Derived Class

```
private: a  
private: b  
c : no access
```

Constructors and Destructors

Constructors and class initialization

- A Derived class inherits from its Base class
- The Base part of the Derived class MUST be initialized BEFORE the Derived class is initialized
- When a Derived object is created
 - Base class constructor executes then
 - Derived class constructor executes

Constructors and Destructors

Constructors and class initialization

```
class Base {  
public:  
    Base() { cout << "Base constructor" << endl; }  
};  
  
class Derived : public Base {  
public:  
    Derived() { cout << "Derived constructor " << endl; }  
};
```

Constructors and Destructors

Constructors and class initialization

Output

Base base;

Base constructor

Derived derived;

Base constructor
Derived constructor

Constructors and Destructors

Destructors

- Class destructors are invoked in the reverse order as constructors
- The Derived part of the Derived class MUST be destroyed BEFORE the Base class destructor is invoked
- When a Derived object is destroyed
 - Derived class destructor executes then
 - Base class destructor executes
 - Each destructor should free resources allocated in it's own constructors

Constructors and Destructors

Destructors

```
class Base {  
public:  
    Base() { cout << "Base constructor" << endl; }  
    ~Base() { cout << "Base destructor" << endl; }  
};  
  
class Derived : public Base {  
public:  
    Derived() { cout << "Derived constructor " << endl; }  
    ~Derived() { cout << "Derived destructor " << endl; }  
};
```

Constructors and Destructors

Destructors and class initialization

Output

Base base;

Base constructor
Base destructor

Derived derived;

Base constructor
Derived constructor
Derived destructor
Base destructor

Constructors and Destructors

Constructors and class initialization

- A Derived class does NOT inherit
 - Base class constructors
 - Base class destructor
 - Base class overloaded assignment operators
 - Base class friend functions
- However, the derived class constructors, destructors, and overloaded assignment operators can invoke the base-class versions
- C++11 allows explicit inheritance of base ‘non-special’ constructors with
 - `using Base::Base;` anywhere in the derived class declaration
 - Lots of rules involved, often better to define constructors yourself

Inheritance

Passing arguments to base class constructors

- The Base part of a Derived class must be initialized first
- How can we control exactly which Base class constructor is used during initialization?
- We can invoke the whichever Base class constructor we wish in the initialization list of the Derived class

Inheritance

Passing arguments to base class constructors

```
class Base {  
public:  
    Base();  
Base(int);  
    . . .  
};
```

```
Derived::Derived(int x)  
: Base(x), {optional initializers for Derived} {  
    // code  
}
```

Constructors and Destructors

Constructors and class initialization

```
class Base {  
    int value;  
public:  
    Base() : value{0} {  
        cout << "Base no-args constructor" << endl;  
    }  
    Base(int x) : value{x} {  
        cout << "int Base constructor" << endl;  
    }  
};
```

Constructors and Destructors

Constructors and class initialization

```
class Derived : public Base {  
    int doubled_value;  
public:  
    Derived() : Base{}, doubled_value{0} {  
        cout << "Derived no-args constructor " << endl;  
    }  
    Derived(int x) : Base{x}, doubled_value {x*2} {  
        cout << "int Derived constructor " << endl;  
    }  
};
```

Constructors and Destructors

Constructors and class initialization

```
Base base;
```

```
Base base{100};
```

```
Derived derived;
```

```
Derived derived{100};
```

Output

Base no-args constructor

int **Base** constructor

Base no-args constructor

Derived no-args constructor

int **Base** constructor

int **Derived** constructor

Inheritance

Copy/Move constructors and overloaded operator=

- Not inherited from the Base class
- You may not need to provide your own
 - Compiler-provided versions may be just fine
- We can explicitly invoke the Base class versions from the Derived class

Inheritance

Copy constructor

- Can invoke Base copy constructor explicitly
 - Derived object '*other*' will be **sliced**

```
Derived::Derived(const Derived &other)
: Base(other), {Derived initialization list}

{
    // code
}
```

Constructors and Destructors

Copy Constructors

```
class Base {  
    int value;  
public:  
    // Same constructors as previous example  
  
    Base(const Base &other) :value{other.value} {  
        cout << "Base copy constructor" << endl;  
    }  
};
```

Constructors and Destructors

Copy Constructors

```
class Derived : public Base {  
    int doubled_value;  
public:  
    // Same constructors as previous example  
  
    Derived(const Derived &other)  
    : Base(other) , doubled_value {other.doubled_value } {  
        cout << "Derived copy constructor " << endl;  
    }  
};
```

Constructors and Destructors

operator=

```
class Base {
    int value;
public:
    // Same constructors as previous example
    Base &operator=(const Base &rhs) {
        if (this != &rhs) {
            value = rhs.value; // assign
        }
        return *this;
    }
};
```

Constructors and Destructors

operator=

```
class Derived : public Base {
    int doubled_value;
public:
    // Same constructors as previous example
    Derived &operator=(const Derived &rhs) {
        if (this != &rhs) {
            Base::operator=(rhs);          // Assign Base part
            doubled_value = rhs.doubled_value; // Assign Derived part
        }
        return *this;
    }
};
```

Inheritance

Copy/Move constructors and overloaded operator=

- Often you do not need to provide your own
- If you **DO NOT** define them in Derived
 - then the compiler will create them and automatically call the base class's version
- If you **DO** provide Derived versions
 - then **YOU** must invoke the Base versions **explicitly** yourself
- Be careful with raw pointers
 - Especially if Base and Derived each have raw pointers
 - Provide them with deep copy semantics

Inheritance

Using and redefining Base class methods

- Derived class can directly invoke Base class methods
- Derived class can **override** or **redefine** Base class methods
- Very powerful in the context of polymorphism
(next section)

Inheritance

Using and redefining Base class methods

```
class Account {  
public:  
    void deposit(double amount) { balance += amount; }  
};
```

```
class Savings_Account: public Account {  
public:  
    void deposit(double amount) { // Redefine Base class method  
        amount += some_interest;  
        Account::deposit(amount); // invoke call Base class method  
    }  
};
```

Inheritance

Static binding of method calls

- Binding of which method to use is done at compile time
 - Default binding for C++ is static
 - Derived class objects will use Derived::deposit
 - But, we can explicitly invoke Base::deposit from Derived::deposit
 - OK, but limited – much more powerful approach is dynamic binding which we will see in the next section

Inheritance

Static binding of method calls

```
Base b;  
b.deposit(1000.0);           // Base::deposit
```

```
Derived d;  
d.deposit(1000.0);           // Derived::deposit
```

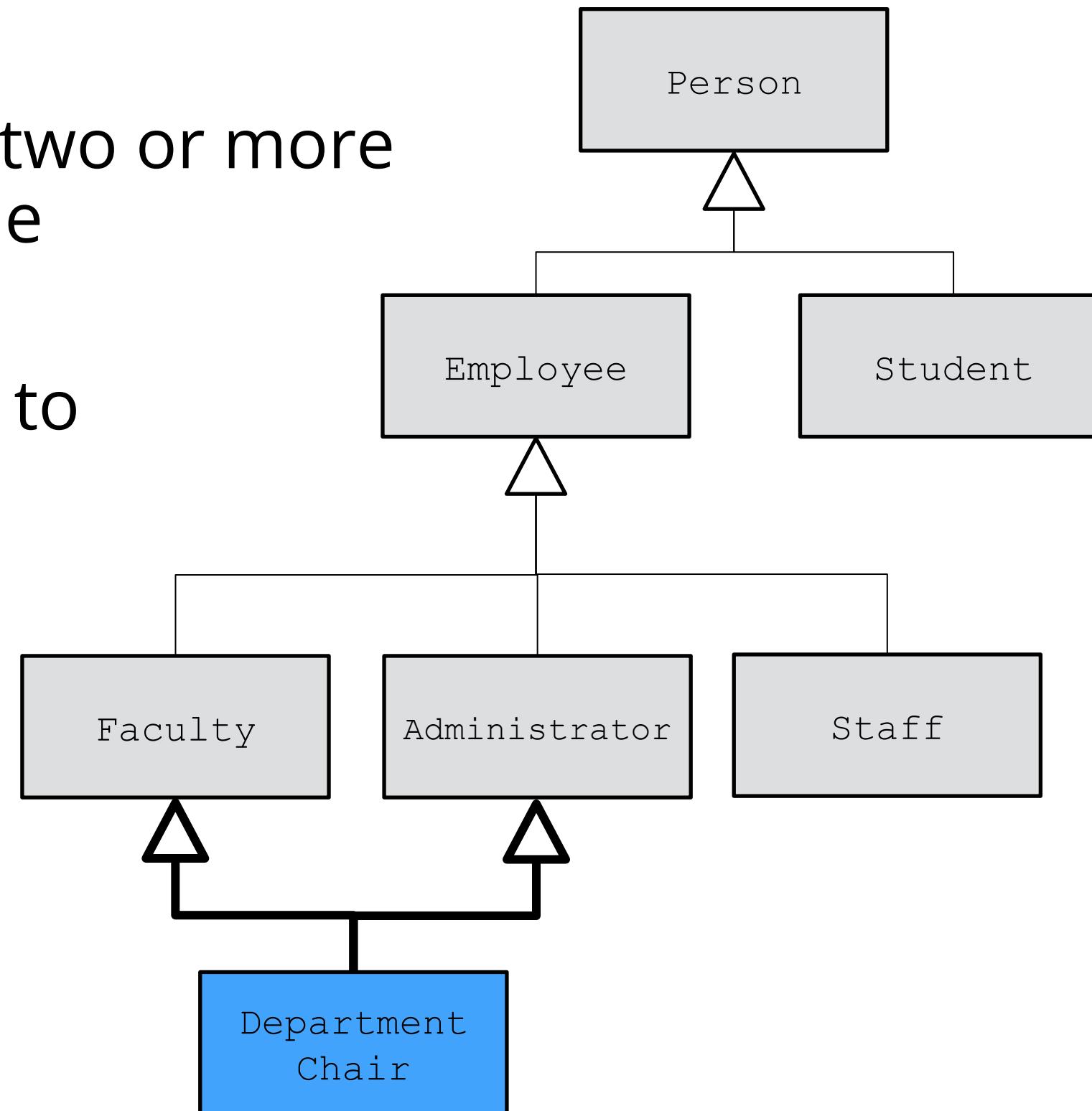
```
Base *ptr = new Derived();  
ptr->deposit(1000.0);        // Base::deposit ????
```

Multiple Inheritance

- A derived class inherits from two or more Base classes at the same time

- The Base classes may belong to unrelated class hierarchies

- A Department Chair
 - Is-A Faculty and
 - Is-A Administrator



Multiple Inheritance

C++ Syntax

```
class Department_Chair:  
    public Faculty, public Administrator {  
    . . .  
};
```

- Beyond the scope of this course
- Some compelling use-cases
- Easily misused
- Can be very complex

Section Overview

Polymorphism and Inheritance

- What is Polymorphism?
- Using base class pointers
- Static vs. dynamic binding
- Virtual functions
- Virtual destructors
- The override and final specifiers
- Using base class references
- Pure virtual functions and abstract classes
- Abstract classes as interfaces

Section Overview

Polymorphism and Inheritance

- What is Polymorphism?
- Using base class pointers
- Static vs. dynamic binding
- Virtual functions
- Virtual destructors
- The override and final specifiers
- Using base class references
- Pure virtual functions and abstract classes
- Abstract classes as interfaces

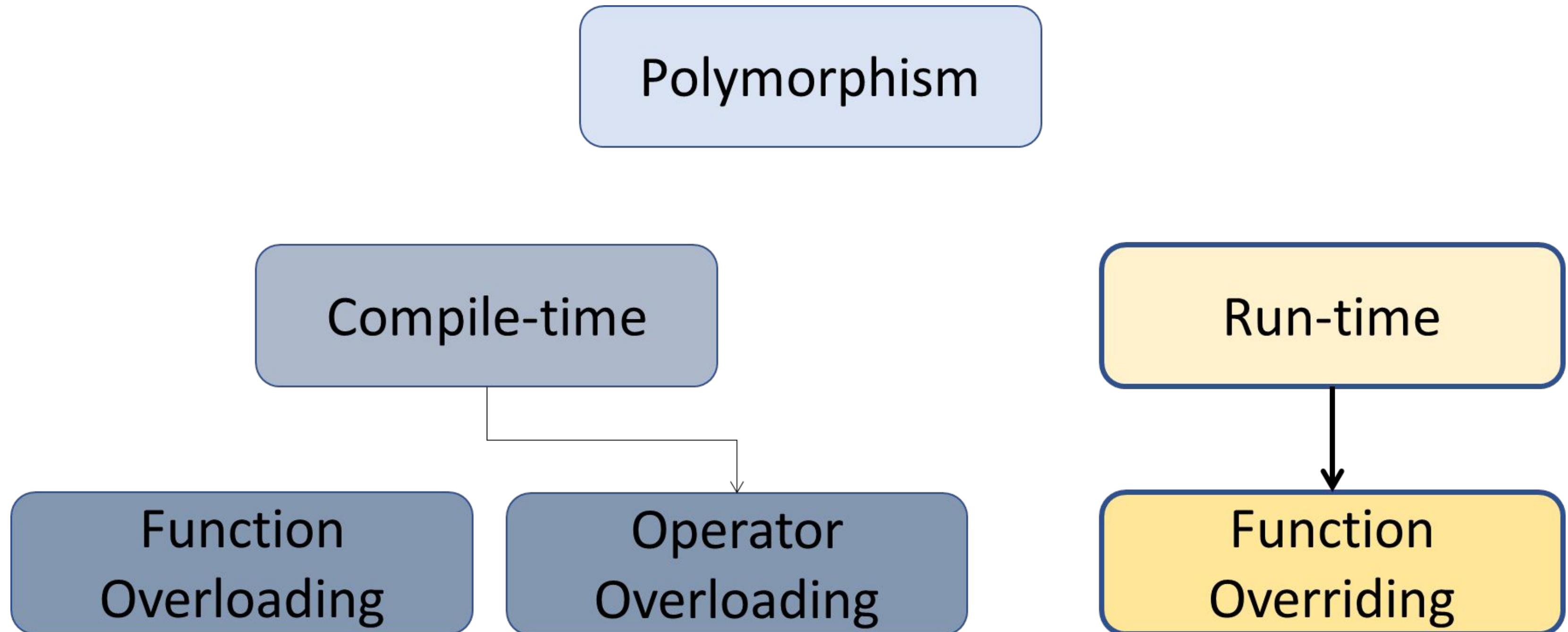
Polymorphism

What is Polymorphism?

- Fundamental to Object-Oriented Programming
- Polymorphism
 - Compile-time / early binding / static binding
 - **Run-time / late binding / dynamic binding**
- Runtime polymorphism
 - Being able to assign different meanings to the same function at run-time
- Allows us to program more abstractly
 - Think general vs. specific
 - Let C++ figure out which function to call at run-time
- Not the default in C++, run-time polymorphism is achieved via
 - Inheritance
 - Base class pointers or references
 - virtual functions

Polymorphism

Types of Polymorphism in C++?



Polymorphism

An non-polymorphic example – Static Binding

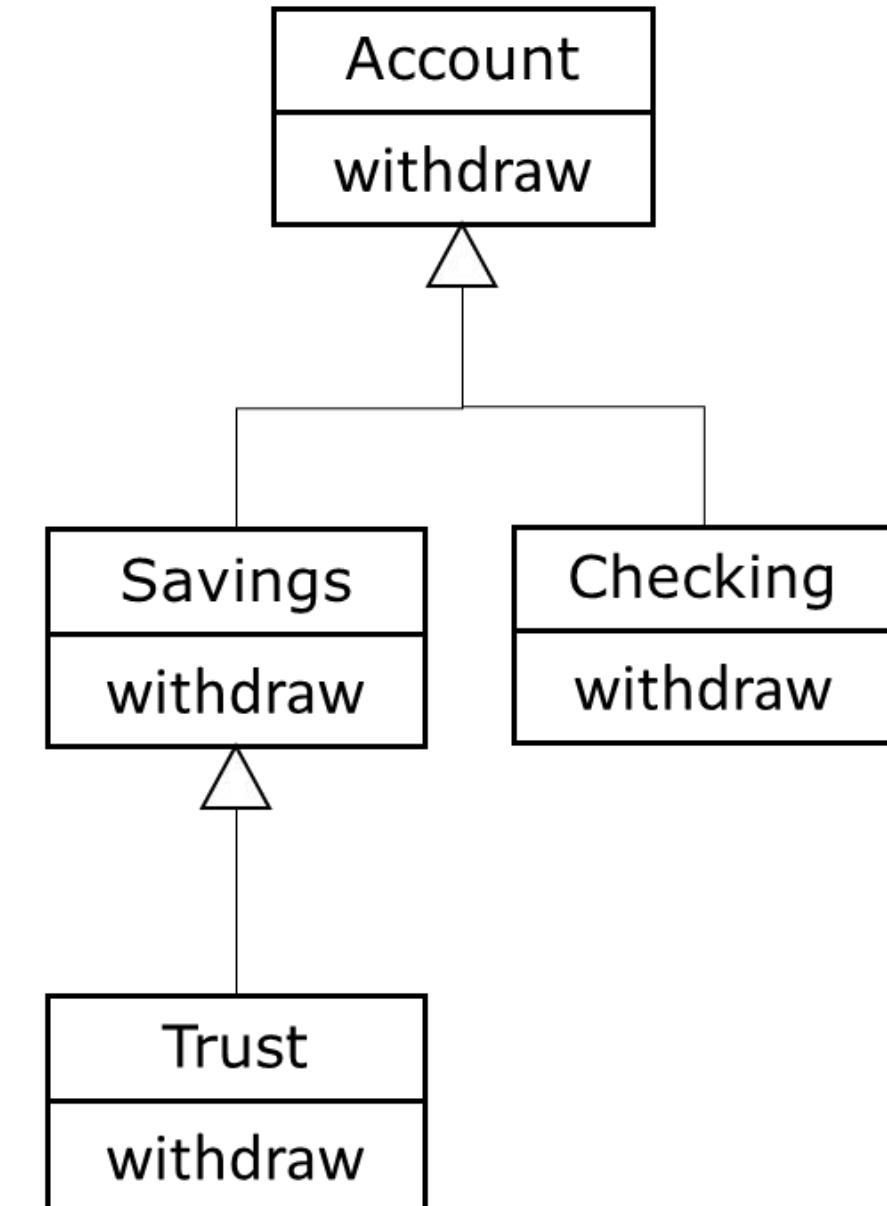
```
Account a;  
a.withdraw(1000); // Account::withdraw()
```

```
Savings b;  
b.withdraw(1000); // Savings::withdraw()
```

```
Checking c;  
c.withdraw(1000); // Checking::withdraw()
```

```
Trust d;  
d.withdraw(1000); // Trust::withdraw()
```

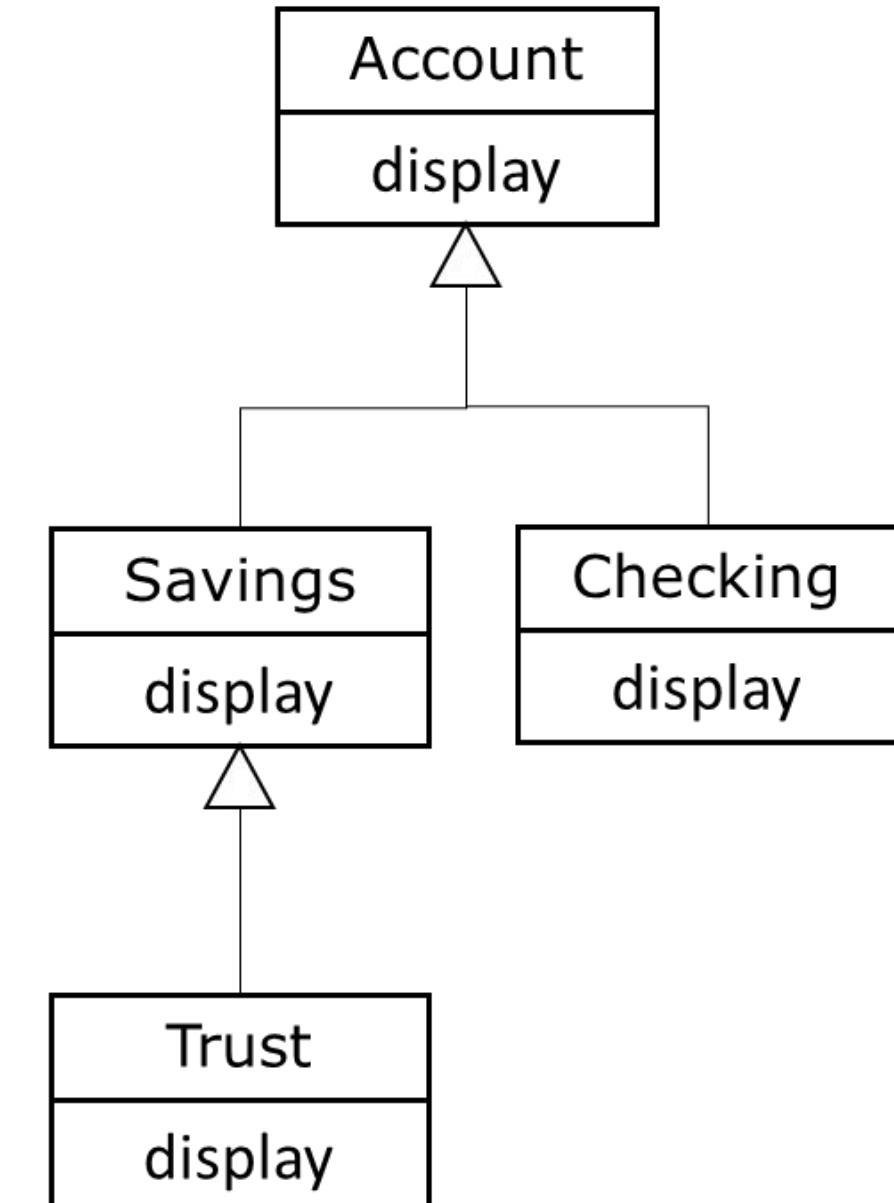
```
Account *p = new Trust();  
p->withdraw(1000); // Account::withdraw()  
// should be  
// Trust::withdraw()
```



Polymorphism

An non-polymorphic example – Static Binding

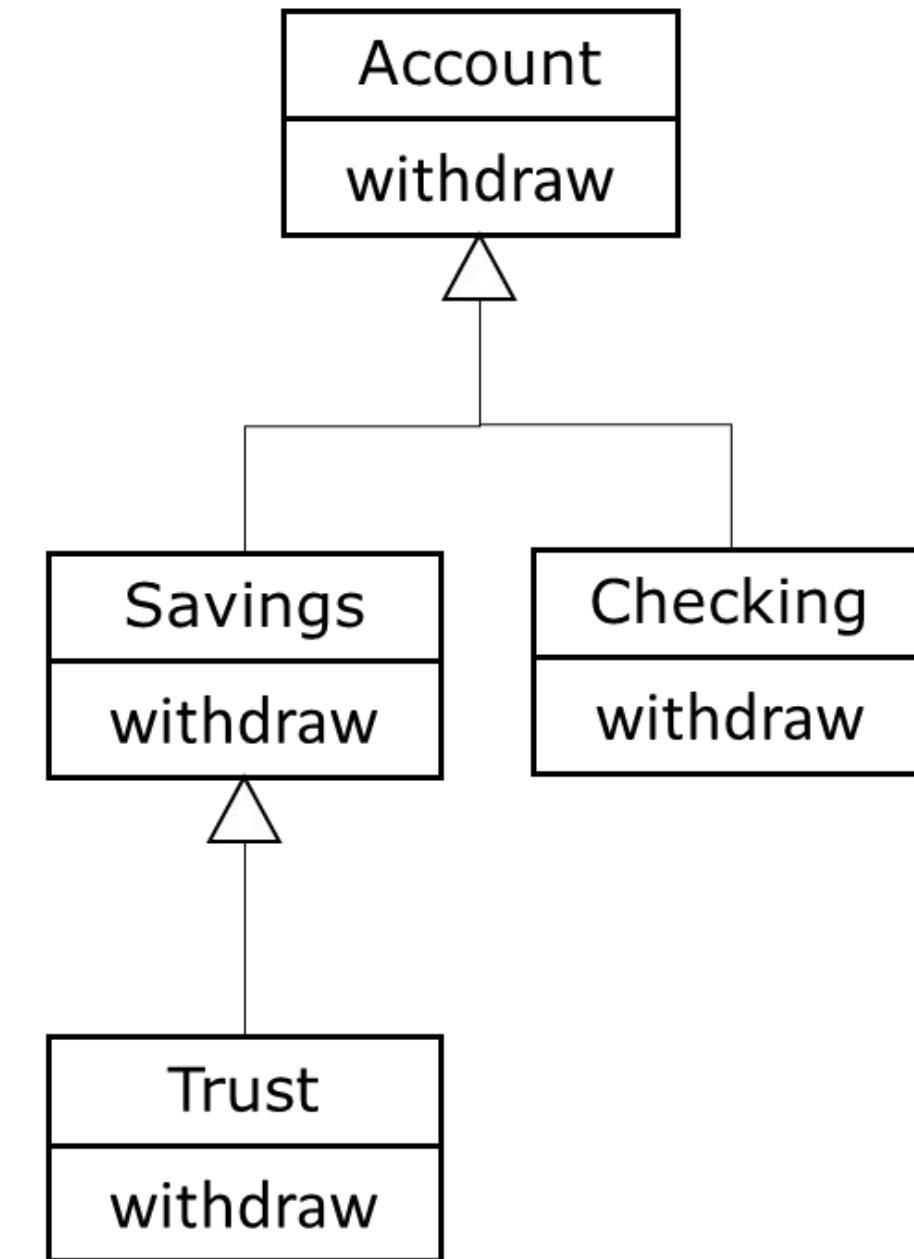
```
void display_account(const Account &acc) {  
    acc.display();  
    // will always use Account::display  
}  
  
Account a;  
display_account(a);  
  
Savings b;  
display_account(b);  
  
Checking c;  
display_account(c);  
  
Trust d;  
display_account(d);
```



Polymorphism

A polymorphic example - Dynamic Binding

```
Account a;  
a.withdraw(1000); // Account::withdraw()  
  
Savings b;  
b.withdraw(1000); // Savings::withdraw()  
  
Checking c;  
c.withdraw(1000); // Checking::withdraw()  
  
Trust d;  
d.withdraw(1000); // Trust::withdraw()  
  
Account *p = new Trust();  
P->withdraw(1000); // Trust::withdraw()
```



withdraw method is virtual in Account

Polymorphism

A polymorphic example - Dynamic Binding

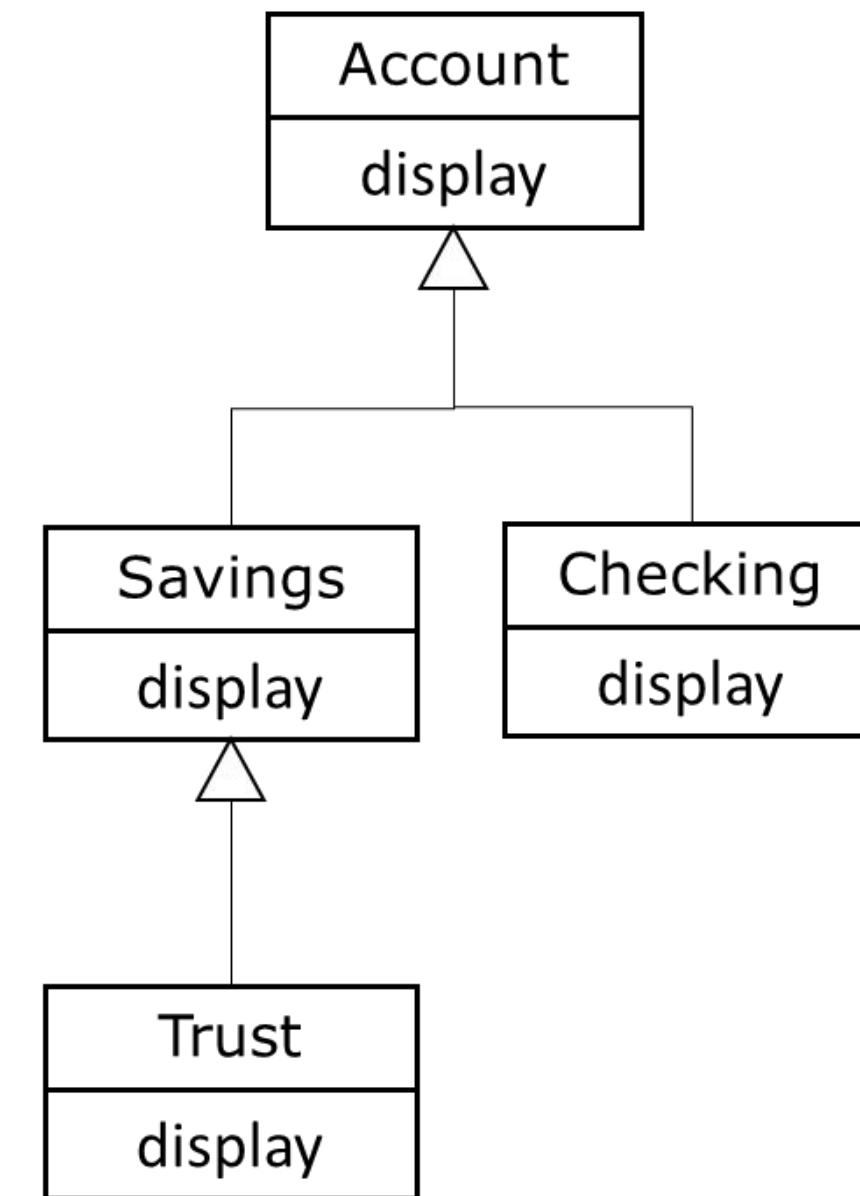
```
void display_account(const Account &acc)
{
    acc.display();
    // will always call the display method
    // depending on the object's type
    // at RUN-TIME!
}

Account a;
display_account(a);

Savings b;
display_account(b);

Checking c;
display_account(c);

Trust d;
display_account(d);
```



display method is virtual in Account

Polymorphism

Using a Base class pointer

- For dynamic polymorphism we must have:

- Inheritance
- Base class pointer or Base class reference
- virtual functions

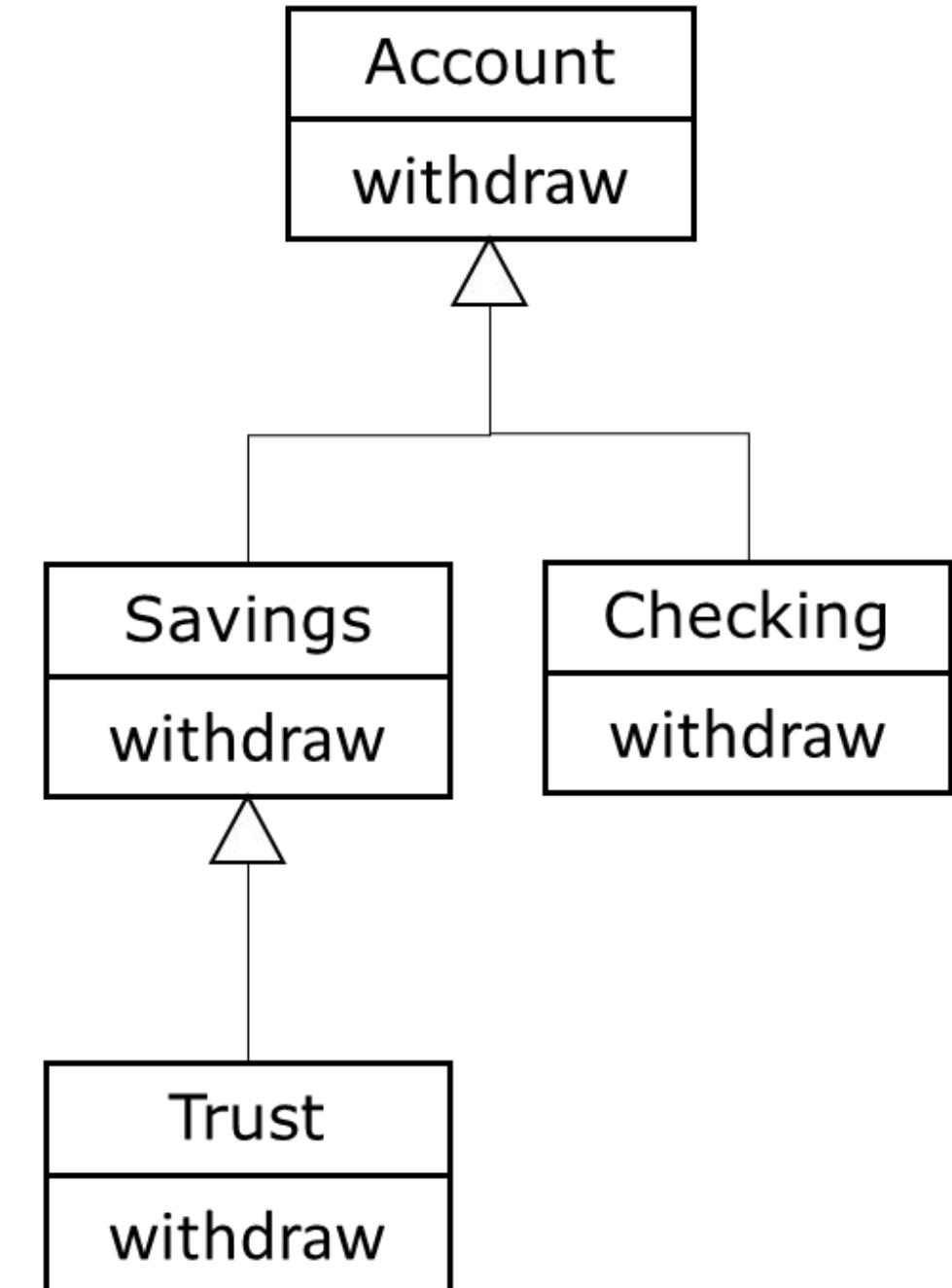
Polymorphism

Using a Base class pointer

```
Account *p1 = new Account();
Account *p2 = new Savings();
Account *p3 = new Checking();
Account *p4 = new Trust();

p1->withdraw(1000);           //Account::withdraw
p2->withdraw(1000);           //Savings::withdraw
p3->withdraw(1000);           //Checking::withdraw
p4->withdraw(1000);           //Trust::withdraw

// delete the pointers
```



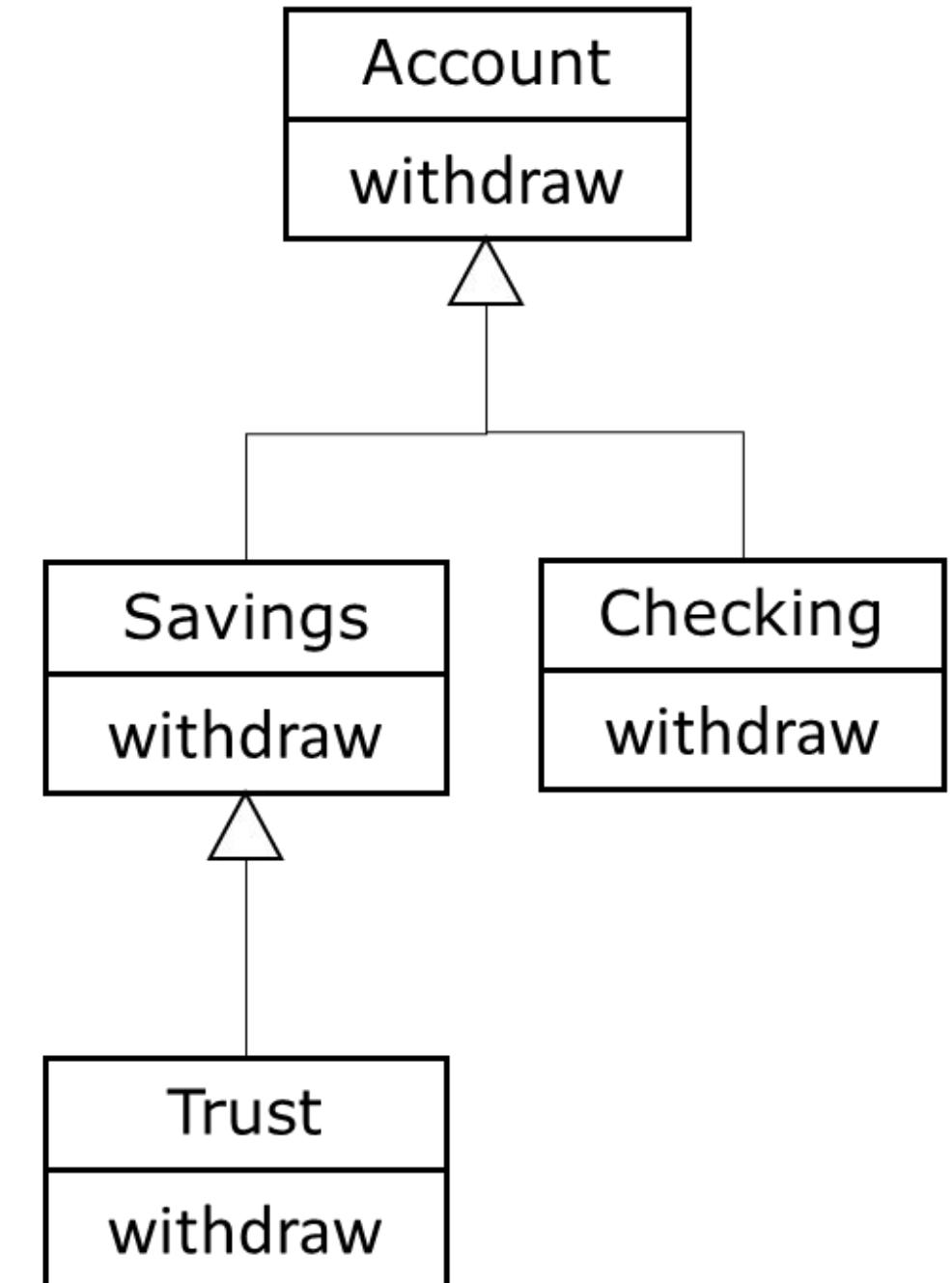
Polymorphism

Using a Base class pointer

```
Account *p1 = new Account();
Account *p2 = new Savings();
Account *p3 = new Checking();
Account *p4 = new Trust();

Account *array [] = {p1,p2,p3,p4};

for (auto i=0; i<4; ++i)
    array[i]->withdraw(1000);
```



Polymorphism

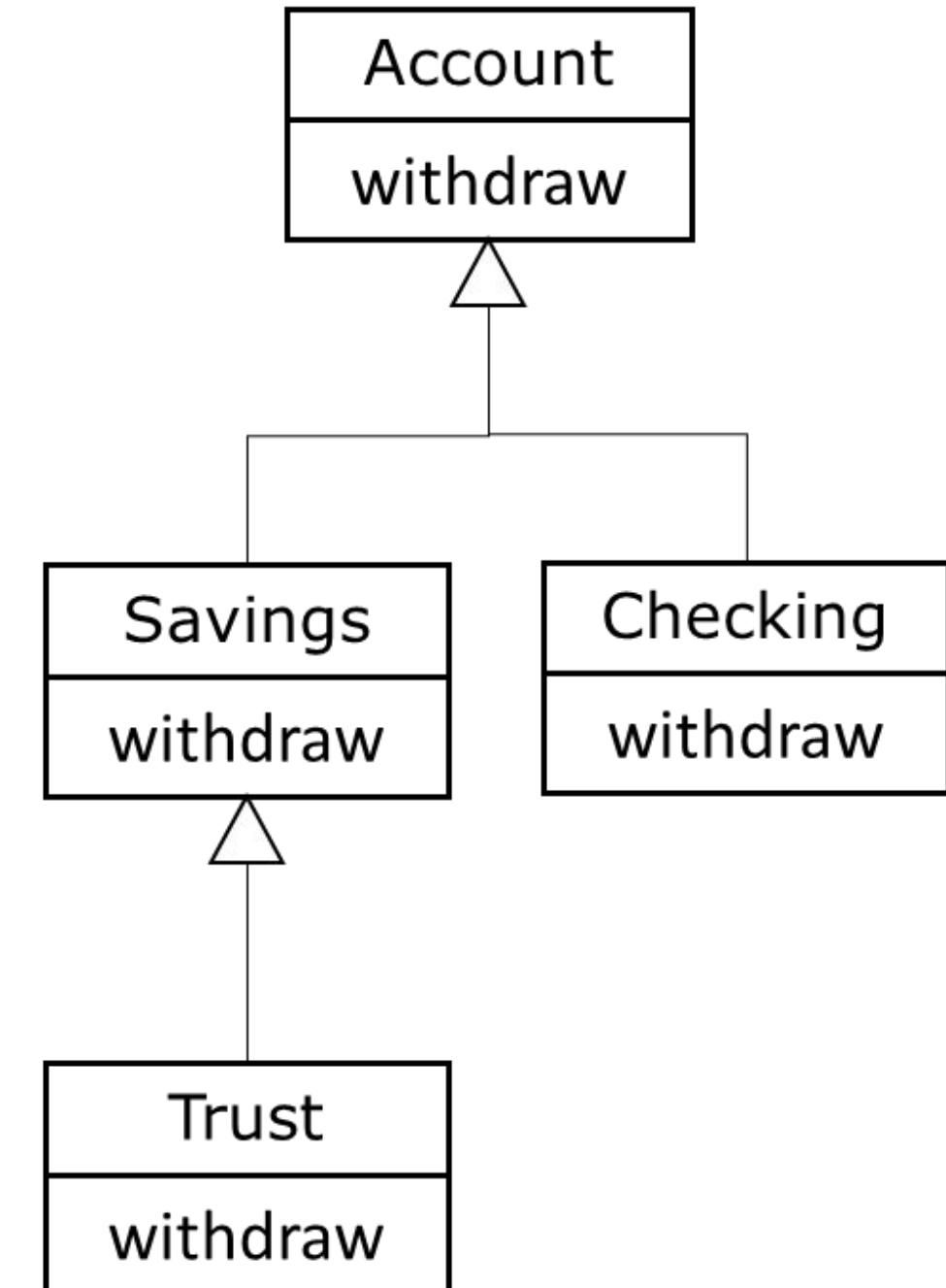
Using a Base class pointer

```
Account *p1 = new Account();
Account *p2 = new Savings();
Account *p3 = new Checking();
Account *p4 = new Trust();

vector<Account *> accounts
{p1, p2, p3, p4};

for (auto acc_ptr: accounts)
    acc_ptr->withdraw(1000);

// delete the pointers
```



Polymorphism

Virtual functions

- Redefined functions are bound statically
- Overridden functions are bound dynamically
- Virtual functions are overridden
- Allow us to treat all objects generally as objects of the Base class

Polymorphism

Declaring virtual functions

- Declare the function you want to override as virtual in the Base class
- Virtual functions are virtual all the way down the hierarchy from this point
- Dynamic polymorphism only via Account class pointer or reference

```
class Account {  
public:  
    virtual void withdraw(double amount);  
    . . .  
};
```

Polymorphism

Declaring virtual functions

- Override the function in the Derived classes
- Function signature and return type must match EXACTLY
- Virtual keyword not required but is best practice
- If you don't provide an overridden version it is inherited from its base class

```
class Checking : public Account {  
public:  
    virtual void withdraw(double amount);  
    . . .  
};
```

Polymorphism

Virtual Destructors

- Problems can happen when we destroy polymorphic objects
- If a derived class is destroyed by deleting its storage via the base class pointer and the class has a non-virtual destructor. Then the behavior is undefined in the C++ standard.
- Derived objects must be destroyed in the correct order starting at the correct destructor

Polymorphism

Virtual Destructors

- Solution/Rule:
 - If a class has virtual functions
 - ALWAYS provide a public virtual destructor
 - If base class destructor is virtual then all derived class destructors are also virtual

```
class Account {  
public:  
    virtual void withdraw(double amount);  
    virtual ~Account();  
    . . .  
};
```

Polymorphism

The override specifier

- We can override Base class virtual functions
- The function signature and return must be EXACTLY the same
- If they are different then we have redefinition NOT overriding
- Redefinition is statically bound
- Overriding is dynamically bound
- C++11 provides an override specifier to have the compiler ensure overriding

Polymorphism

The override specifier

```
class Base {
public:
    virtual void say_hello() const {
        std::cout << "Hello - I'm a Base class object" << std::endl;
    }
    virtual ~Base() { }
};

class Derived: public Base {
public:
    virtual void say_hello() { // Notice I forgot the const - NOT OVERRIDING
        std::cout << "Hello - I'm a Derived class object" << std::endl;
    }
    virtual ~Derived() { }
};
```

Polymorphism

The override specifier

Base:

```
virtual void say_hello() const;
```

Derived:

```
virtual void say_hello();
```

Polymorphism

The override specifier

```
Base *p1 = new Base();  
p1->say_hello(); // "Hello - I'm a Base class object"
```

```
Base *p2 = new Derived();  
p2->say_hello(); // "Hello - I'm a Base class object"
```

- Not what we expected
- say_hello method signatures are different
- So Derived **redefines** say_hello instead of overriding it!

Polymorphism

The override specifier

```
class Base {
public:
    virtual void say_hello() const {
        std::cout << "Hello - I'm a Base class object" << std::endl;
    }
    virtual ~Base() {}
};

class Derived: public Base {
public:
    virtual void say_hello() override { // Produces compiler error
        // Error: marked override but does not override
        std::cout << "Hello - I'm a Derived class object" << std::endl;
    }
    virtual ~Derived() {}
};
```

Polymorphism

The `final` specifier

- C++11 provides the `final` specifier
 - When used at the class level
 - Prevents a class from being derived from
- When used at the method level
- Prevents virtual method from being overridden in derived classes

Polymorphism

final class

```
class My_class final {  
    . . .  
};
```

```
class Derived final: public Base {  
    . . .  
};
```

Polymorphism

final method

```
class A {  
public:  
    virtual void do_something();  
};  
  
class B: public A {  
    virtual void do_something() final; // prevent futher overriding  
};  
  
class C: public B {  
    virtual void do_something(); // COMPILER ERROR - Can't override  
};
```

Polymorphism

Using Base class references

- We can also use Base class references with dynamic polymorphism
- Useful when we pass objects to functions that expect a Base class reference

Polymorphism

Using Base class references

```
Account a;  
Account &ref = a;  
ref.withdraw(1000); // Account::withdraw
```

```
Trust t;  
Account &ref1 = t;  
ref1.withdraw(1000); // Trust::withdraw
```

Polymorphism

Using Base class references

```
void do_withdraw(Account &account, double amount) {  
    account.withdraw(amount);  
}
```

```
Account a;  
do_withdraw(a, 1000); // Account::withdraw
```

```
Trust t;  
do_withdraw(t, 1000); // Trust::withdraw
```

Polymorphism

Pure virtual functions and abstract classes

- Abstract class

- Cannot instantiate objects
- These classes are used as base classes in inheritance hierarchies
- Often referred to as Abstract Base Classes

- Concrete class

- Used to instantiate objects from
- All their member functions are defined
 - Checking Account, Savings Account
 - Faculty, Staff
 - Enemy, Level Boss

Polymorphism

Pure virtual functions and abstract classes

- Abstract base class
 - Too generic to create objects from
 - Shape, Employee, Account, Player
 - Serves as parent to Derived classes that may have objects
 - Contains at least one pure virtual function

Polymorphism

Pure virtual functions and abstract classes

- Pure virtual function
 - Used to make a class abstract
 - Specified with “=0” in its declaration

```
virtual void function() = 0; // pure virtual function
```

- Typically do not provide implementations

Polymorphism

Pure virtual functions and abstract classes

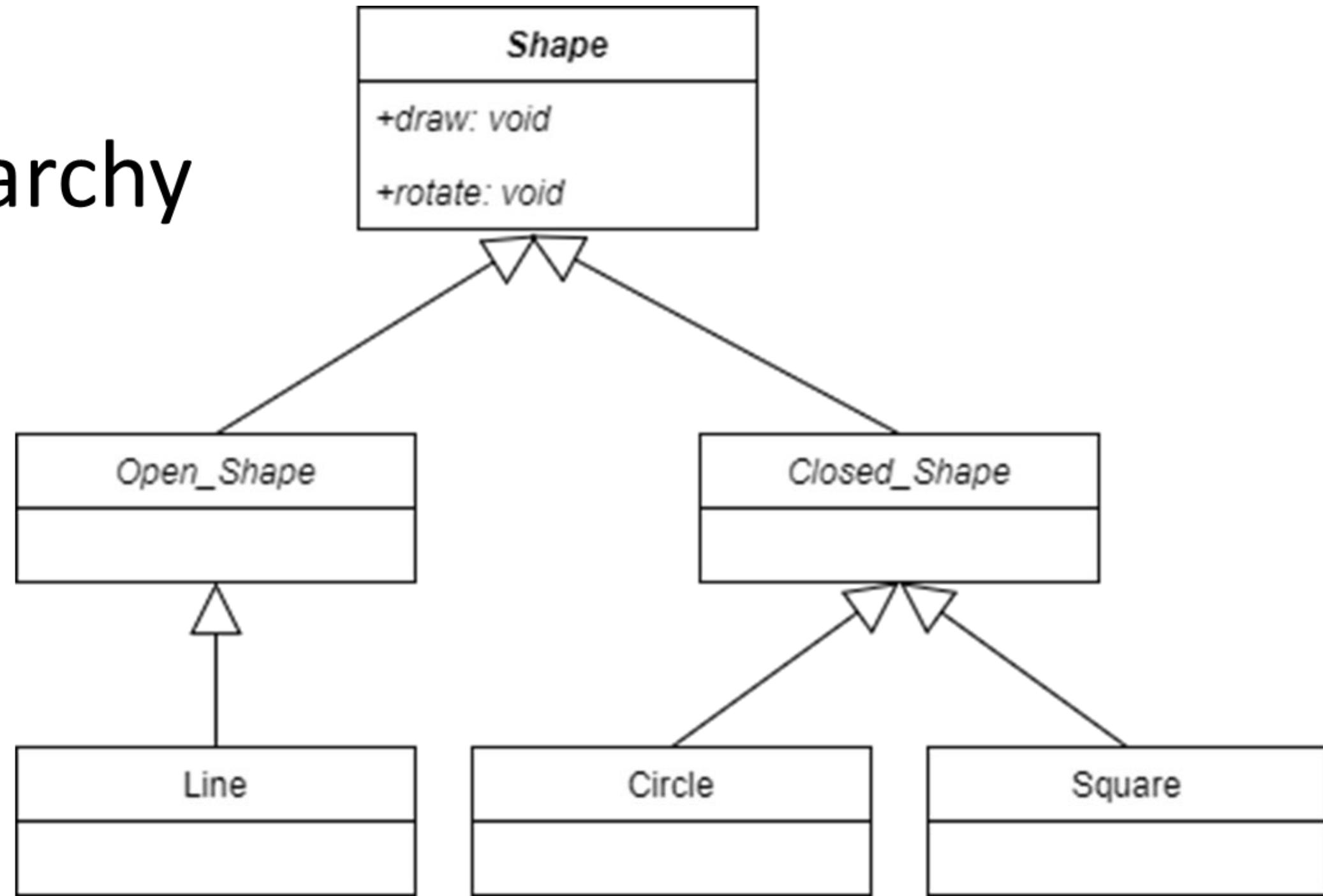
- Pure virtual function
 - Derived classes MUST override the base class
 - If the Derived class does not override then the Derived class is also abstract
 - Used when it doesn't make sense for a base class to have an implementation
 - But concrete classes must implement it

```
virtual void draw() = 0;      // Shape  
virtual void defend() = 0;    // Player
```

Polymorphism

Pure virtual functions and abstract classes

Shape Class Hierarchy



Polymorphism

Pure virtual functions and abstract classes

```
class Shape {                                // Abstract
private:
    // attributes common to all shapes
public:
    virtual void draw() = 0;                  // pure virtual function
    virtual void rotate() = 0;                 // pure virtual function
    virtual ~Shape();
    . . .
};
```

Polymorphism

Pure virtual functions and abstract classes

```
class Circle: public Shape {                      // Concrete
private:
    // attributes for a circle
public:
    virtual void draw() override {
        // code to draw a circle
    }
    virtual void rotate() override {
        // code to rotate a circle
    }
    virtual ~Circle();
    . . .
};
```

Polymorphism

Pure virtual functions and abstract classes

Abstract Base class

- Cannot be instantiated

```
Shape shape;           // Error  
Shape *ptr = new Shape(); // Error
```

- We can use pointers and references to dynamically refer to concrete classes derived from them

```
Shape *ptr = new Circle();  
ptr->draw();  
ptr->rotate();
```

Polymorphism

What is using a class as an interface?

- An abstract class that has only pure virtual functions
- These functions provide a general set of services to the user of the class
- Provided as public
- Each subclass is free to implement these services as needed
- Every service (method) must be implemented
- The service type information is strictly enforced

Polymorphism

A Printable example

- C++ does not provide true interfaces
- We use abstract classes and pure virtual functions to achieve it
- Suppose we want to be able to provide Printable support for any object we wish without knowing its implementation at compile time

```
std::cout << any_object << std::endl;
```

- **any_object** must conform to the Printable interface

Polymorphism

An Printable example

```
class Printable {  
    friend ostream &operator<<(ostream &, const Printable &obj);  
public:  
    virtual void print(ostream &os) const = 0;  
    virtual ~Printable() {};  
    . . .  
};  
  
ostream &operator<<(ostream &os, const Printable &obj) {  
    obj.print(os);  
    return os;  
}
```

Polymorphism

An Printable example

```
class Any_Class : public Printable {  
public:  
    // must override Printable::print()  
    virtual void print(ostream &os) override {  
        os << "Hi from Any_Class" ;  
    }  
    . . .  
};
```

Polymorphism

An Printable example

```
Any_Class *ptr= new Any_Class();
cout << *ptr << endl;

void function1 (Any_Class &obj) {
    cout << obj << endl;
}

void function2 (Printable &obj) {
    cout << obj << endl;
}

function1(*ptr);           // "Hi from Any_Class"
function2(*ptr);           // "Hi from Any_Class"
```

Polymorphism

A Shapes example

```
class Shape {  
public:  
    virtual void draw() = 0;  
    virtual void rotate() = 0;  
    virtual ~Shape() {};  
    . . .  
};
```

Polymorphism

A Shapes example

```
class Circle : public Shape {  
public:  
    virtual void draw() override { /* code */ };  
    virtual void rotate() override { /* code */ };  
    virtual ~Circle() {};  
    . . .  
};
```

Polymorphism

A Shapes example

```
class I_Shape {  
public:  
    virtual void draw() = 0;  
    virtual void rotate() = 0;  
    virtual ~I_Shape() {};  
    . . .  
};
```

Polymorphism

A Shapes example

```
class Circle : public I_Shape {  
public:  
    virtual void draw() override { /* code */ };  
    virtual void rotate() override { /* code */ };  
    virtual ~Circle() {};  
    . . .  
};
```

- Line and Square classes would be similar

Polymorphism

A Shapes example

```
vector< I_Shape * > shapes;

I_Shape *p1 = new Circle();
I_Shape *p2 = new Line();
I_Shape *p3 = new Square();

for (auto const &shape: shapes) {
    shape->rotate();
    shape->draw();
}

// delete the pointers
```

Section Overview

Smart Pointers

- Issues with raw pointers
- What are smart pointers?
- Concept of ownership and RAII
- C++ Smart Pointers
 - Unique pointers (`unique_ptr`)
 - Shared pointers (`shared_ptr`)
 - Weak pointers (`weak_ptr`)
- Custom deleters

Smart Pointers

Issues with Raw Pointers

- C++ provides absolute flexibility with memory management
 - Allocation
 - Deallocation
 - Lifetime management
- Some potentially serious problems
 - Uninitialized (wild) pointers
 - Memory leaks
 - Dangling pointers
 - Not exception safe
- Ownership?
 - Who owns the pointer?
 - When should a pointer be deleted?

Smart Pointers

What are they?

- Objects
- Can only point to heap-allocated memory
- Automatically call delete when no longer needed
- Adhere to RAII principles
- C++ Smart Pointers
 - Unique pointers (`unique_ptr`)
 - Shared pointers (`shared_ptr`)
 - Weak pointers (`weak_ptr`)
 - Auto pointers (`auto_ptr`)

Deprecated – we will not discuss

Smart Pointers

What are they?

- `#include <memory>`
- Defined by class templates
 - Wrapper around a raw pointer
 - Overloaded operators
 - Dereference (*)
 - Member selection (->)
 - Pointer arithmetic not supported (++, --, etc.)
 - Can have custom deleters

Smart Pointers

A simple example

```
{  
    std::smart_pointer<Some_Class> ptr = . . .  
  
    ptr->method();  
    cout << (*ptr) << endl;  
  
}  
  
// ptr will be destroyed automatically when  
// no longer needed
```

Smart Pointers

RAII – Resource Acquisition Is Initialization

- Common idiom or pattern used in software design based on container object lifetime
- RAII objects are allocated on the stack
- Resource Acquisition
 - Open a file
 - Allocate memory
 - Acquire a lock
- Is Initialization
 - The resource is acquired in a constructor
- Resource relinquishing
 - Happens in the destructor
 - Close the file
 - Deallocate the memory
 - Release the lock

Smart Pointers

unique_ptr

- Simple smart pointer – very efficient!
- `unique_ptr<T>`
 - Points to an object of type T on the heap
 - It is unique – there can only be one `unique_ptr<T>` pointing to the object on the heap
 - Owns what it points to
 - Cannot be assigned or copied
 - CAN be moved
 - When the pointer is destroyed, what it points to is automatically destroyed

Smart Pointers

unique_ptr - creating, initializing and using

```
{  
    std::unique_ptr<int> p1 { new int {100} };  
  
    std::cout << *p1 << std::endl;      // 100  
  
    *p1 = 200;  
  
    std::cout << *p1 << std::endl;      // 200  
}  
// automatically deleted
```

Smart Pointers

unique_ptr - some other useful methods

```
{  
    std::unique_ptr<int> p1 {new int {100}};  
  
    std::cout << p1.get() << std::endl; // 0x564388  
  
    p1.reset(); // p1 is now nullptr  
  
    if (p1)  
        std::cout << *p1 << std::endl; // won't execute  
    } // automatically deleted
```

Smart Pointers

unique_ptr - user defined classes

```
{  
    std::unique_ptr<Account> p1 { new Account{"Larry"} };  
    std::cout << *p1 << std::endl; // display account  
  
    p1->deposit(1000);  
    p1->withdraw(500);  
  
} // automatically deleted
```

Smart Pointers

unique_ptr - vectors and move

```
{  
    std::vector<std::unique_ptr<int>> vec;  
  
    std::unique_ptr<int> ptr { new int{100} };  
  
    vec.push_back(ptr); // Error - copy not allowed  
  
    vec.push_back(std::move(ptr));  
  
} // automatically deleted
```

Smart Pointers

unique_ptr - make_unique (C++14)

```
{  
    std::unique_ptr<int> p1 = make_unique<int>(100);  
  
    std::unique_ptr<Account> p2 = make_unique<Account>"Curly", 5000);  
  
auto p3 = make_unique<Player>"Hero", 100, 100);  
  
} // automatically deleted
```

More efficient – no calls to new or delete

Smart Pointers

shared_ptr

- Provides shared ownership of heap objects
- shared_ptr<T>
 - Points to an object of type T on the heap
 - It is not unique – there can many shared_ptrs pointing to the same object on the heap
 - Establishes shared ownership relationship
 - CAN be assigned and copied
 - CAN be moved
 - Doesn't support managing arrays by default
 - When the use count is zero, the managed object on the heap is destroyed

Smart Pointers

shared_ptr - creating, initializing and using

```
{  
    std::shared_ptr<int> p1 { new int {100} };  
  
    std::cout << *p1 << std::endl;      // 100  
  
    *p1 = 200;  
  
    std::cout << *p1 << std::endl;      // 200  
  
} // automatically deleted
```

Smart Pointers

shared_ptr - some other useful methods

```
{  
    // use_count - the number of shared_ptr objects managing the heap object  
    std::shared_ptr<int> p1 {new int {100}};  
    std::cout << p1.use_count () << std::endl;           // 1  
  
    std::shared_ptr<int> p2 { p1 };                      // shared ownership  
    std::cout << p1.use_count () << std::endl;           // 2  
  
    p1.reset();           // decrement the use_count; p1 is nulled out  
    std::cout << p1.use_count() << std::endl;            // 0  
    std::cout << p2.use_count() << std::endl;            // 1  
} // automatically deleted
```

Smart Pointers

shared_ptr - user defined classes

```
{  
    std::shared_ptr<Account> p1 {new Account{"Larry"} } ;  
    std::cout << *p1 << std::endl; // display account  
  
    p1->deposit(1000) ;  
    p1->withdraw(500) ;  
  
} // automatically deleted
```

Smart Pointers

shared_ptr - vectors and move

```
{  
    std::vector<std::shared_ptr<int>> vec;  
  
    std::shared_ptr<int> ptr {new int{100}};  
  
    vec.push_back(ptr); // OK - copy IS allowed  
  
    std::cout << ptr.use_count() << std::endl; // 2  
} // automatically deleted
```

Smart Pointers

shared_ptr - make_shared (C++11)

```
{  
    std::shared_ptr<int> p1 = std::make_shared<int>(100); // use_count: 1  
    std::shared_ptr<int> p2 { p1 }; // use_count : 2  
    std::shared_ptr<int> p3;  
    p3 = p1; // use_count : 3  
  
} // automatically deleted
```

- Use `std::make_shared` – it's more efficient!
- All 3 pointers point to the SAME object on the heap!
- When the `use_count` becomes 0 the heap object is deallocated

Smart Pointers

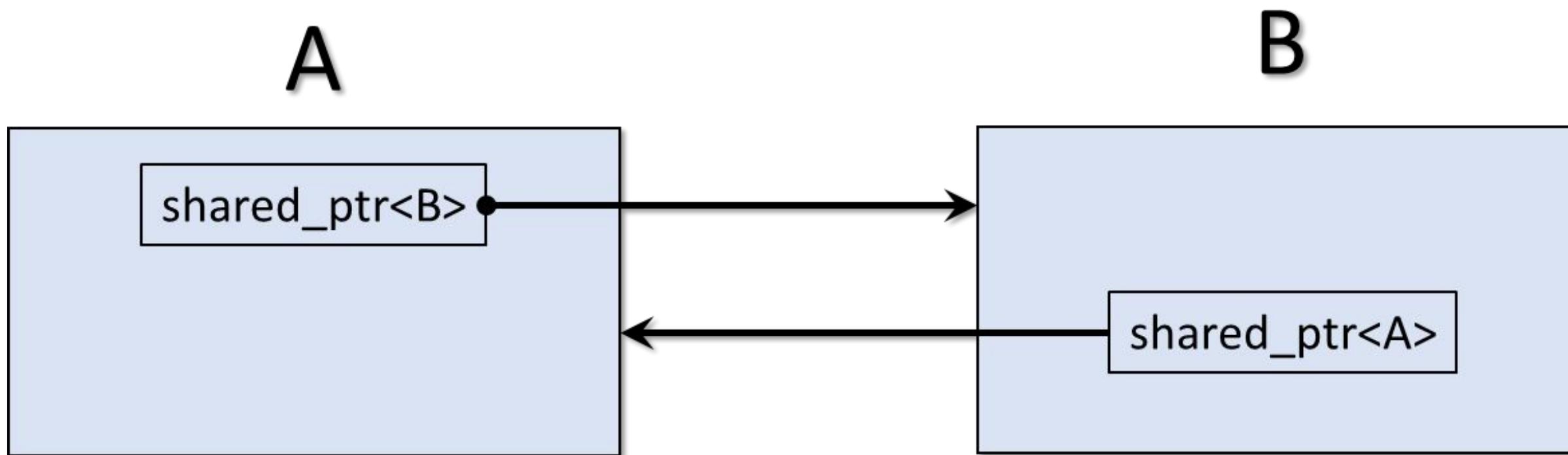
`weak_ptr`

- Provides a non-owning “weak” reference
- `weak_ptr<T>`
 - Points to an object of type T on the heap
 - Does not participate in owning relationship
 - Always created from a `shared_ptr`
 - Does NOT increment or decrement reference use count
 - Used to prevent strong reference cycles which could prevent objects from being deleted

Smart Pointers

`weak_ptr` – circular or cyclic reference

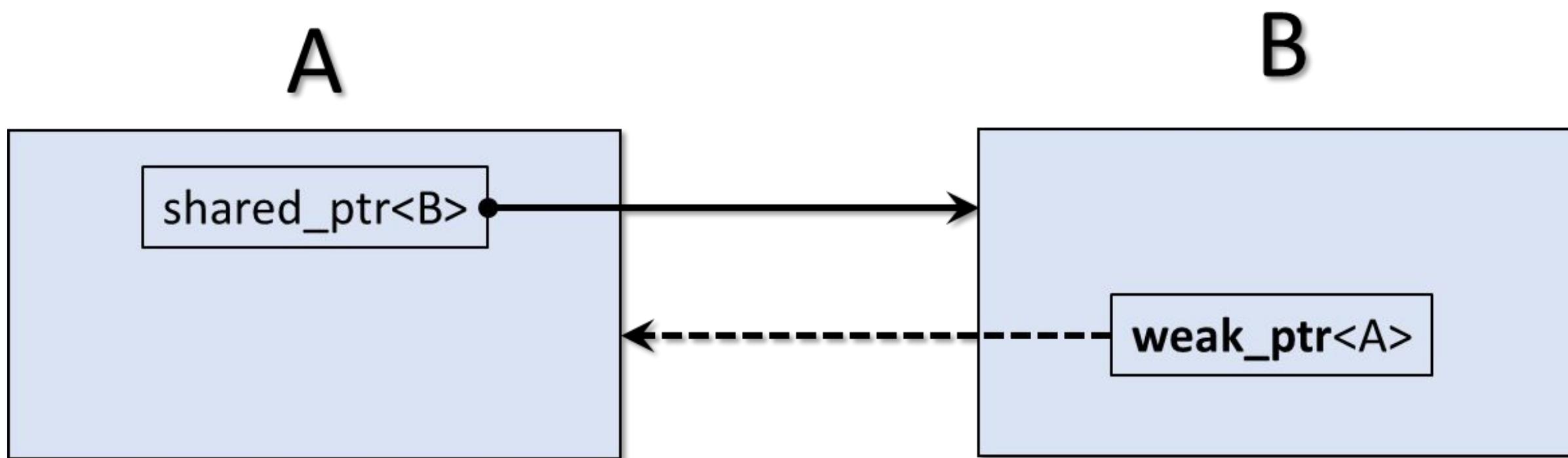
- A refers to B
- B refers to A
- Shared strong ownership prevents heap deallocation



Smart Pointers

`weak_ptr` – circular or cyclic reference

- Solution – make one of the pointers non-owning or ‘weak’
- Now heap storage is deallocated properly



Smart Pointers

Custom deleters

- Sometimes when we destroy a smart pointer we need more than to just destroy the object on the heap
- These are special use-cases
- C++ smart pointers allow you to provide custom deleters
- Lots of way to achieve this
 - Functions
 - Lambdas
 - Others. . .

Smart Pointers

Custom deleters - function

```
void my_deleter(Some_Class *raw_pointer) {
    // your custom deleter code
    delete raw_pointer;
}
```

```
shared_ptr<Some_Class> ptr { new Some_class{}, my_deleter };
```

Smart Pointers

Custom deleters - function

```
void my_deleter(Test *ptr) {  
    cout << "In my custom deleter" << endl;  
    delete ptr;  
}  
  
shared_ptr<Test> ptr { new Test{}, my_deleter };
```

Smart Pointers

Custom deleters - lambda

```
shared_ptr<Test> ptr (new Test{100}, [] (Test *ptr) {  
    cout << "\tUsing my custom deleter" << endl;  
    delete ptr;  
}) ;
```

Section Overview

Exception Handling

- What is an Exception?
- What is Exception Handling?
- What do we throw and catch exceptions?
- How does it affect flow of control?
- Defining our own exception classes
- The Standard Library Exception Hierarchy
 - `std::exception` and `what()`

Exception Handling

Basic concepts

- Exception handling
 - dealing with extraordinary situations
 - indicates that an extraordinary situation has been detected or has occurred
 - program can deal with the extraordinary situations in a suitable manner
- What causes exceptions?
 - insufficient resources
 - missing resources
 - invalid operations
 - range violations
 - underflows and overflows
 - Illegal data and many others
- Exception safe
 - when your code handles exceptions

Exception Handling

Terminology

- Exception
 - an object or primitive type that signals that an error has occurred
- Throwing an exception (raising an exception)
 - your code detects that an error has occurred or will occur
 - the place where the error occurred may not know how to handle the error
 - code can throw an exception describing the error to another part of the program that knows how to handle the error
- Catching an exception (handle the exception)
 - code that handles the exception
 - may or may not cause the program to terminate

Exception Handling

C++ Syntax

- `throw`
 - throws an exception
 - followed by an argument
- `try { code that may throw an exception }`
 - you place code that may throw an exception in a try block
 - if the code throws an exception the try block is exited
 - the thrown exception is handled by a catch handler
 - if no catch handler exists the program terminates
- `catch (Exception ex) { code to handle the exception }`
 - code that handles the exception
 - can have multiple catch handlers
 - may or may not cause the program to terminate

Exception Handling

Divide by zero example

- What happens if `total` is zero?
 - crash, overflow?
 - it depends

```
double average { } ;  
average = sum / total;
```

Exception Handling

Divide by zero example

- What happens if `total` is zero?
 - crash, overflow?
 - it depends

```
double average { } ;  
if (total == 0)  
    // what to do?  
else  
    average = sum / total;
```

Exception Handling

Divide by zero example

```
double average () {
    try {                                // try block
        if (total == 0)
            throw 0;                      // throw the exception
        average = sum / total;           // won't execute if total == 0
        // use average here
    }
    catch (int &ex) {                    // exception handler
        std::cerr << "can't divide by zero" << std::endl;
    }
    std::cout << "program continues" << std::endl;
}
```

Exception Handling

Throwing an exception from a function

What do we return if total is zero?

```
double calculate_avg(int sum, int total) {  
    return static_cast<double>(sum) / total;  
}
```

Exception Handling

Throwing an exception from a function

Throw an exception if we can't complete successfully

```
double calculate_avg(int sum, int total) {  
    if (total == 0)  
        throw 0;  
  
    return static_cast<double>(sum) / total;  
}
```

Exception Handling

Catching an exception thrown from a function

```
double average { };

try {
    average = calculate_avg(sum, total);
    std::cout << average << std::endl;
}

catch (int &ex) {
    std::cerr << "You can't divide by zero" << std::endl;
}

std::cout << "Bye" << std::endl;
```

Exception Handling

Throwing multiple exceptions from a function

What if a function can fail in several ways

- gallons **is zero**
- miles **or** gallons **is negative**

```
double calculate_mpg(int miles, int gallons) {  
    return static_cast<double>(miles) / gallons;  
}
```

Exception Handling

Throwing an exception from a function

Throw different type exceptions for each condition

```
double calculate_mpg(int miles, int gallons) {  
    if (gallons == 0)  
        throw 0;  
    if (miles < 0 || gallons < 0)  
        throw std::string("Negative value error");  
  
    return static_cast<double>(miles) / gallons;  
}
```

Exception Handling

Catching an exception thrown from a function

```
double miles_per_gallon { };
try {
    miles_per_gallon = calculate_mpg(miles, gallons);
    std::cout << miles_per_gallon << std::endl;
}
catch (int &ex) {
    std::cerr << "You can't divide by zero" << std::endl;
}
catch (std::string &ex) {
    std::cerr << ex << std::endl;
}

std::cout << "Bye" << std::endl;
```

Exception Handling

Catching any type of exception

```
catch (int &ex) {  
}  
  
catch (std::string &ex) {  
}  
  
catch (...) {  
    std::cerr << "Unknown exception" << std::endl;  
}
```

Exception Handling

Stack unwinding

If an exception is thrown but not caught in the current scope

C++ tries to find a handler for the exception by unwinding the stack

- Function in which the exception was not caught terminates and is removed from the call stack
- If a try block was used then catch blocks are checked for a match
- If no try block was used or the catch handler doesn't match stack unwinding occurs again
- If the stack is unwound back to main and no catch handler handles the exception the program terminates

Exception Handling

User-defined exceptions

We can create exception classes and throw instances of those classes

Best practice:

- throw an object not a primitive type
- throw an object by value
- catch an object by reference (or const reference)

Exception Handling

Creating exception classes

```
class DivideByZeroException {  
};
```

```
class NegativeValueException {  
};
```

Exception Handling

Throwing user-defined exception classes

```
double calculate_mpg(int miles, int gallons) {  
  
    if (gallons == 0)  
        throw DivideByZeroException();  
    if (miles < 0 || gallons < 0)  
        throw NegativeValueException();  
  
    return static_cast<double>(miles) / gallons;  
}
```

Exception Handling

Catching user-defined exceptions

```
try {
    miles_per_gallon = calculate_mpg(miles, gallons);
    std::cout << miles_per_gallon << std::endl;
}

catch (const DivideByZeroException &ex) {
    std::cerr << "You can't divide by zero" << std::endl;
}

catch (const NegativeValueException &ex) {
    std::cerr << "Negative values aren't allowed" << std::endl;
}

std::cout << "Bye" << std::endl;
```

Exception Handling

Class-level exceptions

Exceptions can also be thrown from within a class:

- Method
 - These work the same way as they do for functions as we've seen
- Constructor
 - Constructors may fail
 - Constructors do not return any value
 - Throw an exception in the constructor if you cannot initialize an object
- Destructor
 - Do NOT throw exceptions from your destructor

Exception Handling

Class-level exceptions

```
Account::Account(std::string name, double balance)
    : name{name}, balance{balance} {
    if (balance < 0.0)
        throw IllegalBalanceException{};  
}
```

Exception Handling

Class-level exceptions

```
try {
    std::unique_ptr<Account> moes_account =
        std::make_unique<Checking_Account>("Moe", -10.0);
    // use moes_account
}
catch (const IllegalBalanceException &ex) {
    std::cerr << "Couldn't create account" << std::endl;
}
```

Exception Handling

The C++ standard library exception class hierarchy

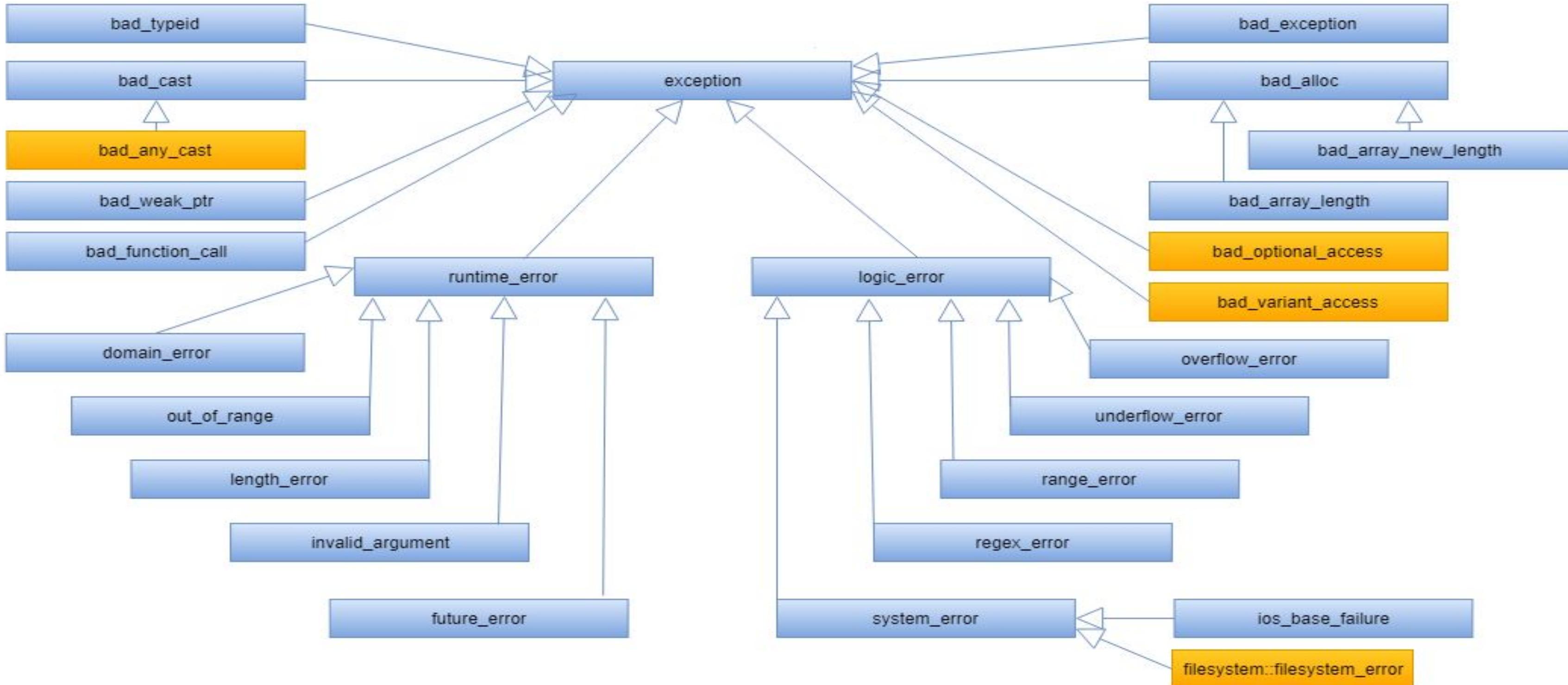
C++ provides a class hierarchy of exception classes

- `std::exception` is the base class
- all subclasses implement the `what()` virtual function
- we can create our own user-defined exception subclasses

```
virtual const char *what() const noexcept;
```

Exception Handling

The C++ standard library exception class hierarchy



Exception Handling

Deriving our class from std::exception

```
class IllegalBalanceException: public std::exception
{
public:
    IllegalBalanceException() noexcept = default;
    ~IllegalBalanceException() = default;
    virtual const char* what() const noexcept {
        return "Illegal balance exception";
    }
};
```

Exception Handling

Our modified Account class constructor

```
Account::Account(std::string name, double balance)
    : name{name}, balance{balance} {
    if (balance < 0.0)
        throw IllegalBalanceException{};
}
```

Exception Handling

Creating an Account object

```
try {
    std::unique_ptr<Account> moes_account =
        std::make_unique<Checking_Account>("Moe", -100.0);

    std::cout << "Use moes_account" << std::endl;
}
catch (const IllegalBalanceException &ex)
{
    std::cerr << ex.what() << std::endl;
    // displays "Illegal balance exception"
}
```

Section Overview

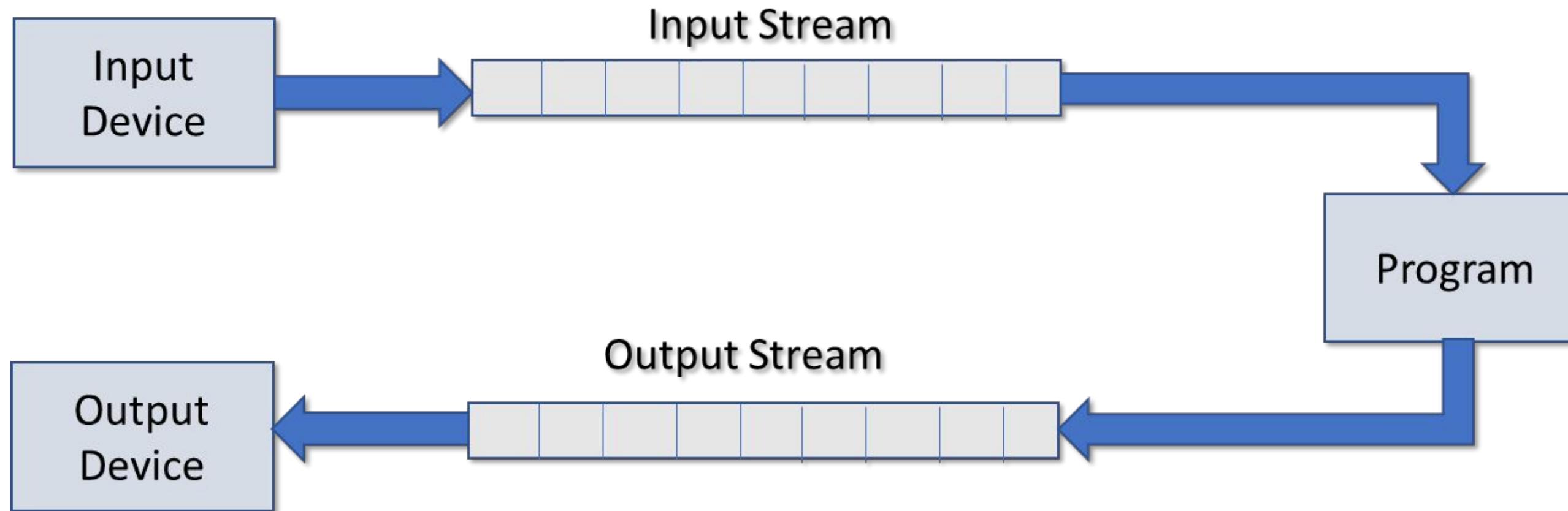
I/O and Streams

- Streams and I/O
- Stream manipulators
- Reading and writing to a text file
- Using string streams

Files, Streams and I/O

- C++ uses streams as an interface between the program and input and output devices
- Independent of the actual device
- Sequence of bytes
- Input stream provides data to the program
- Output stream receives data from the program

Files, Streams and I/O



Files, Streams and I/O

Common header files

Header File	Description
iostream	Provides definitions for formatted input and output from/to streams
fstream	Provides definitions for formatted input and output from/to file streams
iomanip	Provides definitions for manipulators used to format stream I/O

Files, Streams and I/O

Commonly used stream classes

Class	Description
ios	Provides basic support for formatted and unformatted I/O operations. Base class most other classes
ifstream	Provides for high-level input operations on file based streams
ofstream	Provides for high-level output operations on file based streams
fstream	Provides for high-level I/O operations on file based streams Derived from ofstream and ifstream
stringstream	Provides for high-level I/O operations on memory based strings Derived from istringstream and ostringstream

Files, Streams and I/O

Global stream objects

Object	Description
<code>cin</code>	Standard input stream – by default ‘connected’ to the standard input device (keyboard) instance of <code>istream</code>
<code>cout</code>	Standard output stream – by default ‘connected’ to the standard output device (console) instance of <code>ostream</code>
<code>cerr</code>	Standard error stream – by default ‘connected’ to the standard error device (console) Instance of <code>ostream</code> (unbuffered)
<code>clog</code>	Standard log stream – by default ‘connected’ to the standard log device (console) Instance of <code>ostream</code> (unbuffered)

- Global objects – initialized before main executes
- Best practice is to use `cerr` for error messages and `clog` for log messages.

Files, Streams and I/O

Stream manipulators

- Streams have useful member functions to control formatting
- Can be used on input and output streams
- The time of the effect on the stream varies
- Can be used as member functions or as a manipulator

```
std::cout.width(10);           // member function  
std::cout << std::setw(10); // manipulator
```

- We'll focus on manipulator usage

Files, Streams and I/O

Common stream manipulators

- Boolean
 - boolalpha , noboolalpha
- Integer
 - dec, hex, oct, showbase, noshowbase, showpos, noshowpos, uppercase, nouppercase
- Floating point
 - fixed, scientific, setprecision, showpoint, noshowpoint, showpos, noshowpos
- Field width, justification and fill
 - setw, left, right, internal, setfill
- Others
 - endl, flush, skipws, noskipws, ws

Files, Streams and I/O

Formatting boolean types

- Default when displaying boolean values is 1 or 0
- Sometimes the strings true or false are more appropriate

Files, Streams and I/O

Formatting boolean types

```
std::cout << (10 == 10) << std::endl;  
std::cout << (10 == 20) << std::endl;  
  
// Will display  
  
1  
0
```

Files, Streams and I/O

Formatting boolean types

```
std::cout << std::boolalpha;
```

```
std::cout << (10 == 10) << std::endl;  
std::cout << (10 == 20) << std::endl;
```

// Will display

```
true  
false
```

Files, Streams and I/O

Formatting boolean types

- All further boolean output will be affected

```
std::cout << std::noboolalpha; // 1 or 0
```

```
std::cout << std::boolalpha; // true or false
```

Files, Streams and I/O

Formatting boolean types

- Method version

```
std::cout.setf(std::ios::boolalpha);
```

```
std::cout.setf(std::ios::noboolalpha);
```

- Reset to default

```
std::cout << std::resetiosflags(std::ios::boolalpha);
```

Files, Streams and I/O

Formatting integer types

- Default when displaying integer values is:
 - dec (base 10)
 - noshowbase - prefix used to show hexadecimal or octal
 - nouppercase - when displaying a prefix and hex values it will be lower case
 - noshowpos – no '+' is displayed for positive numbers
- These manipulators affect all further output to the stream

Files, Streams and I/O

Formatting integer types - setting base

```
int num {255};  
  
std::cout << std::dec << num << std::endl;  
std::cout << std::hex << num << std::endl;  
std::cout << std::oct << num << std::endl;  
  
// Will display  
255  
ff  
377
```

Files, Streams and I/O

Formatting integer types - showing the base

```
int num {255};

std::cout << std::showbase;      // std::noshowbase
std::cout << std::dec << num << std::endl;
std::cout << std::hex << num << std::endl;
std::cout << std::oct << num << std::endl;

// Will display
255
0xff           // note the 0x prefix for hexadecimal
0377           // note the 0 prefix for octal
```

Files, Streams and I/O

Formatting integer types - display hex in uppercase

```
int num {255};  
  
std::cout << std::showbase << std::uppercase;  
std::cout << std::hex << num << std::endl;  
  
// Will display  
  
0xFF // note capitalized XFF
```

Files, Streams and I/O

Formatting integer types - displaying the positive sign

```
int num1 {255};  
int num2 {-255};  
  
std::cout << num1 << std::endl;      // 255  
std::cout << num2 << std::endl;      // -255  
  
std::cout << std::showpos;           // std::noshowpos  
  
std::cout << num1 << std::endl;      // +255  
std::cout << num2 << std::endl;      // -255
```

Files, Streams and I/O

Setting/resetting integer types

- Set using `setf`

```
std::cout.setf(std::ios::showbase);  
std::cout.setf(std::ios::uppercase);  
std::cout.setf(std::ios::showpos);
```

- Reset to defaults

```
std::cout << std::resetiosflags(std::ios::basefield);  
std::cout << std::resetiosflags(std::ios::showbase);  
std::cout << std::resetiosflags(std::ios::showpos);  
std::cout << std::resetiosflags(std::ios::uppercase);
```

Files, Streams and I/O

Formatting floating point types

- Default when displaying floating point values is:
 - setprecision – number of digits displayed (6)
 - fixed – not fixed to a specific number of digits after the decimal point
 - noshowpoint – trailing zeroes are not displayed
 - nouppercase - when displaying in scientific notation
 - noshowpos – no '+' is displayed for positive numbers
- These manipulators affect all further output to the stream

Files, Streams and I/O

Formatting floating point types - precision

```
double num {1234.5678};  
  
std::cout << num << std::endl;  
  
// Will display  
  
1234.57 // Notice precision is 6 and rounding
```

Files, Streams and I/O

Formatting floating point types - precision

```
double num {123456789.987654321};  
  
std::cout << num << std::endl;  
  
// Will display  
  
1.23457e+008 // Notice precision is 6
```

Files, Streams and I/O

Formatting floating point types - precision

```
double num {123456789.987654321};  
  
std::cout << std::setprecision(9);  
std::cout << num << std::endl;  
  
// Will display  
  
123456790 // Note that rounding occurs
```

Files, Streams and I/O

Formatting floating point types - fixed

```
double num {123456789.987654321};
```

```
std::cout << std::fixed;  
std::cout << num << std::endl;
```

```
// Will display precision 6 from the decimal
```

123456789.**987654**

Files, Streams and I/O

Formatting floating point types - fixed

```
double num {123456789.987654321};  
  
std::cout << std::setprecision(3) << std::fixed;  
std::cout << num << std::endl;  
  
// Will display precision 3 from the decimal
```

123456789.**988**

Files, Streams and I/O

Formatting floating point types - scientific

```
double num {123456789.987654321};
```

```
std::cout << std::setprecision(3)  
      << std::scientific;  
std::cout << num << std::endl;
```

```
// Will display precision 3
```

1.23e+008

Files, Streams and I/O

Formatting floating point types - scientific uppercase

```
double num {123456789.987654321};  
  
std::cout << std::setprecision(3)  
        << std::scientific  
        << std::uppercase;  
std::cout << num << std::endl;  
  
// Will display precision 3  
  
1.23E+008 // Note the capital 'E'
```

Files, Streams and I/O

Formatting floating point types - displaying the positive sign

```
double num {123456789.987654321};

std::cout << std::setprecision(3)
           << fixed
           << std::showpos;

std::cout << num << std::endl;

// Will display

+123456789.988 // Note the leading '+'
```

Files, Streams and I/O

Formatting floating point types - trailing zeroes

```
double num {12.34};

std::cout << num << std::endl;           // 12.34

std::cout << std::showpoint;

std::cout << num << std::endl;           // 12.3400

// Will display

12.34      // Note no trailing zeroes (default)
12.3400    // Note trailing zeroes up to precision
```

Files, Streams and I/O

Returning to general settings

- **unsetf**

```
std::cout.unsetf(std::ios::scientific | std::ios::fixed);
```

or

```
std::cout << std::resetiosflags(std::ios::floatfield);
```

- Refer to the docs for other set/reset flags

Files, Streams and I/O

Field width, align and fill

- Default when displaying floating point values is:
 - setw – not set by default
 - left – when no field width, right - when using field width
 - fill – not set by default – blank space is used
- Some of these manipulators affect only the next data element put on the stream

Files, Streams and I/O

Defaults

```
double num {1234.5678};  
std::string hello{"Hello"};  
  
std::cout << num << hello << std::endl;  
  
// Will display  
  
1234.57Hello
```

Files, Streams and I/O

Defaults

```
double num {1234.5678};  
std::string hello{"Hello"};  
  
std::cout << num << std::endl;  
std::cout << hello << std::endl;  
  
// Will display  
  
1234.57  
Hello
```

Files, Streams and I/O

Field width - setw

```
double num {1234.5678};  
std::string hello{"Hello"};  
  
std::cout << std::setw(10) << num  
      << hello << std::endl;  
  
// Will display  
1234567890123456789012345678901234567890  
1234 . 57Hello
```

Files, Streams and I/O

Field width - setw

```
double num {1234.5678};  
std::string hello{"Hello"};  
  
std::cout << std::setw(10) << num  
      << std::setw(10) << hello  
      << std::setw(10) << hello << std::endl;
```

```
// Will display  
1234567890123456789012345678901234567890  
1234.57      Hello      Hello
```

Files, Streams and I/O

Field width - justification

```
double num {1234.5678};  
std::string hello{"Hello"};
```

```
std::cout << std::setw(10)  
      << std::left  
      << num          // only affects num!  
      << hello << std::endl;
```

```
// Will display  
1234567890123456789012345678901234567890  
1234.57    Hello
```

Files, Streams and I/O

Field width - setw

```
double num {1234.5678};  
std::string hello{"Hello"};
```

```
std::cout << std::setw(10) << num  
        << std::setw(10) << std::right << hello  
        << std::setw(15) << std::right << hello  
        << std::endl;
```

```
// Will display  
1234567890123456789012345678901234567890  
1234.57      Hello          Hello
```

Files, Streams and I/O

Filling fixed width - setfill

```
double num {1234.5678};  
std::string hello{"Hello"};  
  
std::cout << std::setfill('-');  
std::cout << std::setw(10) << num  
    << hello << std::endl;  
  
// Will display  
123456789012345678901234567890  
---1234.57Hello
```

Files, Streams and I/O

Field width - setw

```
double num {1234.5678};  
std::string hello{"Hello"};  
  
std::cout << std::setfill('*');  
std::cout << std::setw(10) << num  
    << std::setfill('-') << std::setw(10) << hello  
    << std::setw(15) << hello  
    << std::endl;  
  
// Will display  
1234567890123456789012345678901234567890  
***1234.57-----Hello-----Hello
```

Files, Streams and I/O

Input files (fstream and ifstream)

`fstream` and `ifstream` are commonly used for input files

1. `#include <fstream>`
2. Declare an `fstream` or `ifstream` object
3. Connect it to a file on your file system (opens it for reading)
4. Read data from the file via the stream
5. Close the stream

Files, Streams and I/O

Opening a file for reading with (fstream)

```
std::fstream in_file {"../myfile.txt",  
                     std::ios::in };
```

- Open for reading in binary mode

```
std::fstream in_file {"../myfile.txt",  
                     std::ios::in | std::ios::binary };
```

Files, Streams and I/O

Opening a file for reading with (ifstream)

```
std::ifstream in_file {"../myfile.txt",  
                      std::ios::in } ;
```

```
std::ifstream in_file {"../myfile.txt"} ;
```

- Open for reading in binary mode

```
std::ifstream in_file {"../myfile.txt",  
                      std::ios::binary} ;
```

Files, Streams and I/O

Opening a file for reading with open

```
std::ifstream in_file;  
std::string filename;  
std::cin >> filename; // get the file name  
  
in_file.open(filename);  
// or  
in_file.open(filename, std::ios::binary);
```

Files, Streams and I/O

Check if file opened successfully (`is_open`)

```
if (in_file.is_open()) {  
    // read from it  
} else {  
    // file could not be opened  
    // does it exist?  
    // should the program terminate?  
}
```

Files, Streams and I/O

Check if file opened successfully - test the stream object

```
if (in_file) {    // just check the object
    // read from it
} else {
    // file could not be opened
    // does it exist?
    // should the program terminate?
}
```

Files, Streams and I/O

Closing a file

- Always close any open files to flush out any unwritten data

```
in_file.close();
```

Files, Streams and I/O

Reading from files using (>>)

- We can use the extraction operator for formatted read
- Same way we used it with cin

```
int num {};  
double total {};  
std::string name {};
```

```
in_file >> num;  
in_file >> total >> name;
```

```
100  
255.67  
Larry
```

Files, Streams and I/O

Reading from files using `getline`

- We can use `getline` to read the file one line at a time

```
std::string line{ };
```

```
std::getline(in_file, line);
```

This is a line

Files, Streams and I/O

Reading text file one line at a time

```
std::ifstream in_file("../myfile.txt");      // open file
std::string line {};

if (!in_file) {          // check if file is open
    std::cerr << "File open error" << std::endl;
    return 1;           // exit the program (main)
}
while (!in_file.eof()) {          // while not at the end
    std::getline(in_file, line);     // read a line
    cout << line << std::endl;     // display the line
}
in_file.close(); // close the file
```

Files, Streams and I/O

Reading text file one line at a time

```
std::ifstream in_file("../myfile.txt"); // open file
std::string line {};

if (!in_file) {      // check if file is open
    std::cerr << "File open error" << std::endl;
    return 1;        // exit the program (main)
}
while (std::getline(in_file, line)) // read a line
    cout << line << std::endl;        // display the line

in_file.close();      // close the file
```

Files, Streams and I/O

Reading text file one character at a time (get)

```
std::ifstream in_file("../myfile.txt"); // open file
char c;

if (!in_file) {           // check if file is open
    std::cerr << "File open error" << std::endl;
    return 1;             // exit the program (main)
}
while (in_file.get(c)) // read a character
    cout << c;          // display the character

in_file.close();         // close the file
```

Files, Streams and I/O

Output files (fstream and ofstream)

`fstream` and `ofstream` are commonly used for output files

1. `#include <fstream>`
2. Declare an `fstream` or `ofstream` object
3. Connect it to a file on your file system (opens it for writing)
4. Write data to the file via the stream
5. Close the stream

Files, Streams and I/O

Output files (fstream and ofstream)

`fstream` and `ofstream` are commonly used for output files

- Output files will be created if they don't exist
- Output files will be overwritten (truncated) by default
- Can be opened so that new writes append
- Can be open in text or binary modes

Files, Streams and I/O

Opening a file for writing with (fstream)

```
std::fstream out_file {"../myfile.txt",  
                         std::ios::out };
```

- Open for writing in binary mode

```
std::fstream out_file {"../myfile.txt",  
                         std::ios::out | std::ios::binary};
```

Files, Streams and I/O

Opening a file for writing with (ofstream)

```
std::ofstream out_file {"../myfile.txt",  
                      std::ios::out};
```

```
std::ofstream out_file {"../myfile.txt"};
```

- Open for writing in binary mode

```
std::ofstream out_file {"../myfile.txt",  
                      std::ios::binary};
```

Files, Streams and I/O

Opening a file for writing with (ofstream)

```
// truncate (discard contents) when opening  
std::ofstream out_file {"./myfile.txt",  
                         std::ios::trunc};
```

```
// append on each write  
std::ofstream out_file {"./myfile.txt",  
                         std::ios::app};
```

```
// seek to end of stream when opening  
std::ofstream out_file {"./myfile.txt",  
                         std::ios::ate};
```

Files, Streams and I/O

Opening a file for writing with open

```
std::ofstream out_file;  
std::string filename;  
std::cin >> filename; // get the file name  
  
out_file.open(filename);  
// or  
out_file.open(filename, std::ios::binary);
```

Files, Streams and I/O

Check if file opened successfully (`is_open`)

```
if (out_file.is_open()) {  
    // read from it  
} else {  
    // file could not be created or opened  
    // does it exist?  
    // should the program terminate?  
}
```

Files, Streams and I/O

Check if file opened successfully - test the stream object

```
if (out_file) { // just check the object
    // read from it
} else {
    // file could not be opened
    // does it exist?
    // should the program terminate?
}
```

Files, Streams and I/O

Closing a file

- Always close any open files to flush out any unwritten data

```
out_file.close();
```

Files, Streams and I/O

Writing to files using (<<)

- We can use the insertion operator for formatted write
- Same way we used it with cout

```
int num {100};  
double total {255.67};  
std::string name {"Larry"};
```

```
out_file << num << "\n"  
    << total << "\n"  
    << name << std::endl;
```

```
100  
255.67  
Larry
```

Files, Streams and I/O

Copying a text file one line at a time

```
std::ifstream in_file("../myfile.txt"); // open file
std::ofstream out_file("../copy.txt");

if (!in_file) {      // check if file is open
    std::cerr << "File open error" << std::endl;
    return 1;        // exit the program (main)
}

if (!out_file) {      // check if file is open
    std::cerr << "File create error" << std::endl;
    return 1;        // exit the program (main)
}
```

Files, Streams and I/O

Copying a text file one line at a time

```
std::string line { };

while (std::getline(in_file, line)) // read a line
    out_file << line << std::endl; // write a line

in_file.close(); // close the files
out_file.close();
```

Files, Streams and I/O

Copying a text file one character at a time (get/put)

```
std::ifstream in_file("../../../myfile.txt"); // open file
std::ofstream out_file("../../../copy.txt");

if (!in_file) {          // check if file is open
    std::cerr << "File open error" << std::endl;
    return 1;             // exit the program (main)
}
if (!out_file) {          // check if file is open
    std::cerr << "File create error" << std::endl;
    return 1;             // exit the program (main)
}
```

Files, Streams and I/O

Copying a text file one character at a time (get/put)

```
char c;  
  
while (in_file.get(c)) // read a character  
    out_file.put(c);      // write the character  
  
in_file.close();        // close the files  
out_file.close();
```

Files, Streams and I/O

Using string streams

- Allow us to read or write from strings in memory much as we would read and write to files
- Very powerful
- Very useful for data validation

Files, Streams and I/O

Using string streams

stringstream, istringstream and ostringstream

1. #include <sstream>
2. Declare an stringstream, istringstream or ostringstream object
3. Connect it to a std::string
4. Read/write data from/to the string stream using formatted I/O

Files, Streams and I/O

Reading from a stringstream

```
#include <sstream>

int num { };
double total { };
std::string name { };
std::string info {"Moe 100 1234.5"};

std::istringstream iss{info};
iss >> name >> num >> total;
```

Files, Streams and I/O

Writing to a stringstream

```
#include <sstream>

int num {100};
double total {1234.5};
std::string name {"Moe"};

std::ostringstream oss {};
oss << name << " " << num << " " << total;
std::cout << oss.str() << std::endl;
```

Files, Streams and I/O

Validating input with stringstream

```
int value {};
std::string input{};

std::cout << "Enter an integer: ";
std::cin >> input;

std::stringstream ss{input};
if (ss >> value) {
    std::cout << "An integer was entered";
} else
    std::cout << "An integer was NOT entered";
```

Files, Streams and I/O

File Locations and IDEs

IDE	File Path
CodeLite	<code>std::ifstream in_file {"../test.txt"};</code>
Windows Visual Studio	<code>std::ifstream in_file {"test.txt"};</code>
Code::Blocks	<code>std::ifstream in_file {"test.txt"};</code>
CLion	<code>std::ifstream in_file {"../test.txt"};</code>
Xcode	See the video

Files, Streams and I/O

File Locations and IDEs

Windows Visual Studio

Files, Streams and I/O

File Locations and IDEs

Code::Blocks

Files, Streams and I/O

File Locations and IDEs

CLion

Files, Streams and I/O

File Locations and IDEs

Xcode

Section Overview

The Standard Template Library

- What is the STL
- Generic programming/
Meta-programming
 - Preprocessor macros
 - Function templates
 - Class templates
- STL Containers
- STL Iterators
- STL Algorithms
- Array
- Vector
- Deque
- List and Forward List
- Set and Multi Set
- Map and Multi Map
- Stack and Queue
- Priority Queue
- Algorithms

What is the STL?

- A library of powerful, reusable, adaptable, generic classes and functions
- Implemented using C++ templates
- Implements common data structures and algorithms
- Huge class library!!
- Alexander Stepanov (1994)

Why use the STL?

- Assortment of commonly used containers
- Known time and size complexity
- Tried and tested – Reusability!!!
- Consistent, fast, and type-safe
- Extensible

Elements of the STL

- Containers

- Collections of objects or primitive types
(array, vector, deque, stack, set, map, etc.)

- Algorithms

- Functions for processing sequences of elements from containers
(find, max, count, accumulate, sort, etc.)

- Iterators

- Generate sequences of element from containers
(forward, reverse, by value, by reference, constant, etc.)

Elements of the STL

A simple example

```
#include <vector>
#include <algorithm>

std::vector<int> v {1, 5, 3};
```

Elements of the STL

A simple example - sort a vector

```
std::sort(v.begin(), v.end());  
  
for (auto elem: v)  
    std::cout << elem << std::endl;
```

1
3
5

Elements of the STL

A simple example - reverse a vector

```
std::reverse(v.begin(), v.end());
```

```
for (auto elem: v)
    std::cout << elem << std::endl;
```

5
3
1

Elements of the STL

A simple example - accumulate

```
int sum{};

sum = std::accumulate(v.begin(), v.end(), 0);
std::cout << sum << std::endl;

9 // 1+3+5
```

Types of Containers

- Sequence containers
 - array, vector, list, forward_list, deque
- Associative containers
 - set, multi set, map, multi map
- Container adapters
 - stack, queue, priority queue

Types of Iterators

- Input iterators – from the container to the program
- Output iterators – from the program to the container
- Forward iterators – navigate one item at a time in one direction
- Bi-directional iterators – navigate one item at a time both directions
- Random access iterators – directly access a container item

Types of Algorithms

- About 60 algorithms in the STL
- Non-modifying
- Modifying

The Standard Template Library

Generic Programming with macros

- Generic programming

“Writing code that works with a variety of types as arguments, as long as those argument types meet specific syntactic and semantic requirements”, Bjarne Stroustrup

- Macros ***** beware *****

- Function templates

- Class templates

The Standard Template Library

Macros (#define)

- C++ preprocessor directives
- No type information
- Simple substitution

```
#define MAX_SIZE 100
```

```
#define PI 3.14159
```

The Standard Template Library

Macros (#define)

```
#define MAX_SIZE 100
```

```
#define PI 3.14159
```

```
if (num > MAX_SIZE)
    std::cout << "Too big";
```

```
double area = PI * r * r;
```

The Standard Template Library

Macros (#define)

```
//#define MAX_SIZE 100      // removed
```

```
//#define PI 3.14159        // removed
```

```
if (num > 100)  
    std::cout << "Too big";
```

```
double area = 3.14159 * r * r;
```

The Standard Template Library

max function

- Suppose we need a function to determine the max of 2 integers

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

```
int x = 100;  
int y = 200;  
std::cout << max(x, y);      // displays 200
```

The Standard Template Library

max function

- Now suppose we need to determine the max of 2 doubles, and 2 chars

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

```
double max(double a, double b) {  
    return (a > b) ? a : b;  
}
```

```
char max(char a, char b) {  
    return (a > b) ? a : b;  
}
```

The Standard Template Library

Macros with arguments (#define)

- We can write a generic macro with arguments instead

```
#define MAX(a, b) ((a > b) ? a : b)
```

```
std::cout << MAX(10,20)      << std::endl;    // 20
std::cout << MAX(2.4, 3.5) << std::endl;    // 3.5
std::cout << MAX('A', 'C') << std::endl;    // C
```

The Standard Template Library

Macros with argument s (#define)

- We have to be careful with macros

```
#define SQUARE (a) a*a
```

```
result = SQUARE(5);           // Expect 25
result = 5*5;                // Get 25
```

```
result = 100/SQUARE(5);      // Expect 4
result = 100/5*5              // Get 100!
```

The Standard Template Library

Macros with argument s (#define)

```
#define SQUARE(a) ((a)*(a)) // note the parenthesis  
  
result = SQUARE(5); // Expect 25  
result = ((5)*(5)); // Still Get 25  
  
result = 100/SQUARE(5); // Expect 4  
result = 100/((5)*(5)); // Now we get 4!!
```

The Standard Template Library

Generic Programming with function templates

What is a C++ Template?

- Blueprint
- Function and class templates
- Allow **plugging-in** any data type
- Compiler generates the appropriate function/class from the blueprint
- Generic programming / meta-programming

The Standard Template Library

Generic Programming with function templates

- Let's revisit the max function from the last lecture

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

```
int x = 100;  
int y = 200;  
std::cout << max(x, y);      // displays 200
```

The Standard Template Library

max function

- Now suppose we need to determine the max of 2 doubles, and 2 chars

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

```
double max(double a, double b) {  
    return (a > b) ? a : b;  
}
```

```
char max(char a, char b) {  
    return (a > b) ? a : b;  
}
```

The Standard Template Library

max function as a template function

- We can replace type we want to generalize with a name, say **T**
- But now this won't compile

```
T max(T a, T b) {  
    return (a > b) ? a : b;  
}
```

The Standard Template Library

max function as a template function

- We need to tell the compiler this is a template function
- We also need to tell it that **T** is the template parameter

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

The Standard Template Library

max function as a template function

- We may also use **class** instead of **typename**

```
template <class T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

The Standard Template Library

max function as a template function

- Now the compiler can generate the appropriate function from the template
- Note, this happens at compile-time!

```
int a {10};  
int b {20};
```

```
std::cout << max<int>(a, b);
```

The Standard Template Library

max function as a template function

- Many times the compiler can deduce the type and the template parameter is not needed
- Depending on the type of a and b, the compiler will figure it out

```
std::cout << max<double>(c, d) ;
```

```
std::cout << max(c, d) ;
```

The Standard Template Library

max function as a template function

- And we can use **almost** any type we need

```
char a { 'A' };  
char b { 'Z' };
```

```
std::cout << max(a, b) << std::endl;
```

The Standard Template Library

max function as a template function

- Notice the type MUST support the `>` operator either natively or as an overloaded operator (`operator>`)

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

The Standard Template Library

max function as a template function

- The following will not compile unless Player overloads operator>

```
Player p1{ "Hero", 100, 20 };
```

```
Player p2{ "Enemy", 99, 3 };
```

```
std::cout << max<Player>(p1, p2) ;
```

The Standard Template Library

multiple types as template parameters

- We can have multiple template parameters
- And their types can be different

```
template <typename T1, typename T2>
void func(T1 a, T2 b) {
    std::cout << a << " " << b;
}
```

The Standard Template Library

multiple types as template parameters

- When we use the function we provide the template parameters
- Often the compiler can deduce them

```
func<int,double>(10, 20.2);
```

```
func('A', 12.4);
```

The Standard Template Library

Generic Programming with class templates

What is a C++ **Class** Template?

- Similar to function template, but at the class level
- Allows **plugging-in** any data type
- Compiler generates the appropriate class from the blueprint

The Standard Template Library

Generic Programming with class templates

- Let's say we want a class to hold Items where the item has a name and an integer

```
class Item {  
private:  
    std::string name;  
    int value;  
public:  
    Item(std::string name, int value)  
        : name{name}, value{value}  
    {}  
    std::string get_name() const {return name; }  
    int get_value() const { return value; }  
};
```

The Standard Template Library

Generic Programming with class templates

- But we'd like our `Item` class to be able to hold any type of data in addition to the string
- We can't overload class names
- We don't want to use dynamic polymorphism

The Standard Template Library

Generic Programming with class templates

```
class Item {  
private:  
    std::string name;  
    T value;  
public:  
    Item(std::string name, T value)  
        : name{name}, value{value}  
    {}  
    std::string get_name() const {return name; }  
    T get_value() const { return value; }  
};
```

The Standard Template Library

Generic Programming with class templates

```
template <typename T>
class Item {
private:
    std::string name;
    T value;
public:
    Item(std::string name, T value)
        : name{name}, value{value}
    {}
    std::string get_name() const {return name; }
    T get_value() const { return value; }
};
```

The Standard Template Library

Generic Programming with class templates

```
Item<int> item1 {"Larry", 1};
```

```
Item<double> item2 {"House", 1000.0};
```

```
Item<std::string> item3 {"Frank", "Boss"};
```

```
std::vector<Item<int>> vec;
```

The Standard Template Library

Multiple types as template parameters

- We can have multiple template parameters
- And their types can be different

```
template <typename T1, typename T2>
struct My_Pair {
    T1 first;
    T2 second;
};
```

The Standard Template Library

Multiple types as template parameters

```
My_Pair <std::string, int> p1 {"Frank", 100};
```

```
My_Pair <int, double> p2 {124, 13.6};
```

```
std::vector<My_Pair<int, double>> vec;
```

The Standard Template Library

std::pair

```
#include <utility>

std::pair<std::string, int> p1 {"Frank", 100};

std::cout << p1.first;           // Frank
std::cout << p1.second;         // 100
```

The Standard Template Library

Containers

- Data structures that can store object of *almost* any type
 - Template-based classes
- Each container has member functions
 - Some are specific to the container
 - Others are available to all containers
- Each container has an associated header file
 - `#include <container_type>`

The Standard Template Library

Containers – common

Function	Description
Default constructor	Initializes an empty container
Overloaded constructors	Initializes containers with many options
Copy constructor	Initializes a container as a copy of another container
Move constructor	Moves existing container to new container
Destructor	Destroys a container
Copy assignment (operator=)	Copy one container to another
Move assignment (operator=)	Move one container to another
size	Returns the number of elements in the container
empty	Returns boolean – is the container empty?
insert	Insert an element into the container

The Standard Template Library

Containers – common

Function	Description
operator< and operator<=	Returns boolean - compare contents of 2 containers
operator> and operator>=	Returns boolean - compare contents of 2 containers
operator== and operator!=	Returns boolean - are the contents of 2 containers equal or not
swap	Swap the elements of 2 containers
erase	Remove element(s) from a container
clear	Remove all elements from a container
begin and end	Returns iterators to first element or end
rbegin and rend	Returns reverse iterators to first element or end
cbegin and cend	Returns constant iterators to first element or end
crbegin and crend	Returns constant reverse iterators to first element or end

The Standard Template Library

Container elements

What types of elements can we store in containers?

- A **copy** of the element will be stored in the container
 - All primitives OK
- Element should be
 - Copyable and assignable (copy constructor / copy assignment)
 - Moveable for efficiency (move Constructor / move Assignment)
- Ordered associative containers must be able to compare elements
 - operator<, operator==

The Standard Template Library

Iterators

- Allows abstracting an arbitrary container as a sequence of elements
- They are objects that work like pointers by design
- Most container classes can be traversed with iterators

The Standard Template Library

Declaring iterators

- Iterators must be declared based on the container type they will iterate over

```
container_type::iterator_type iterator_name;
```

```
std::vector<int>::iterator it1;
```

```
std::list<std::string>::iterator it2;
```

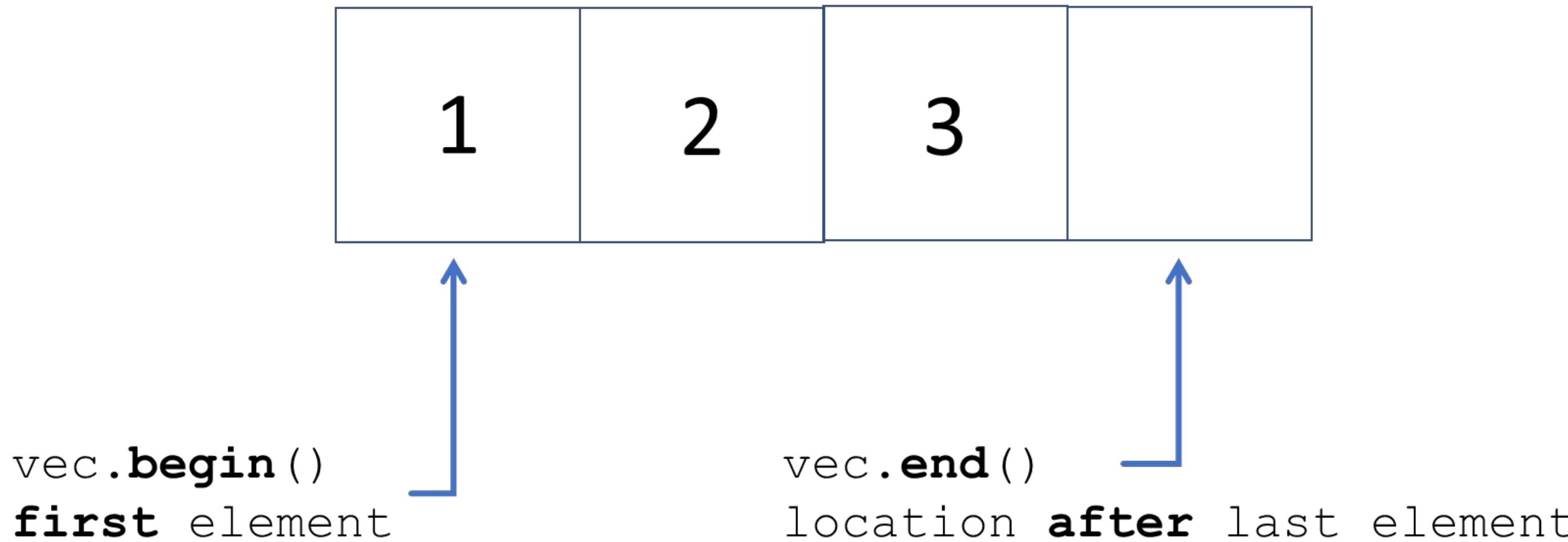
```
std::map<std::string, std::string>::iterator it3;
```

```
std::set<char>::iterator it4;
```

The Standard Template Library

iterator begin and end methods

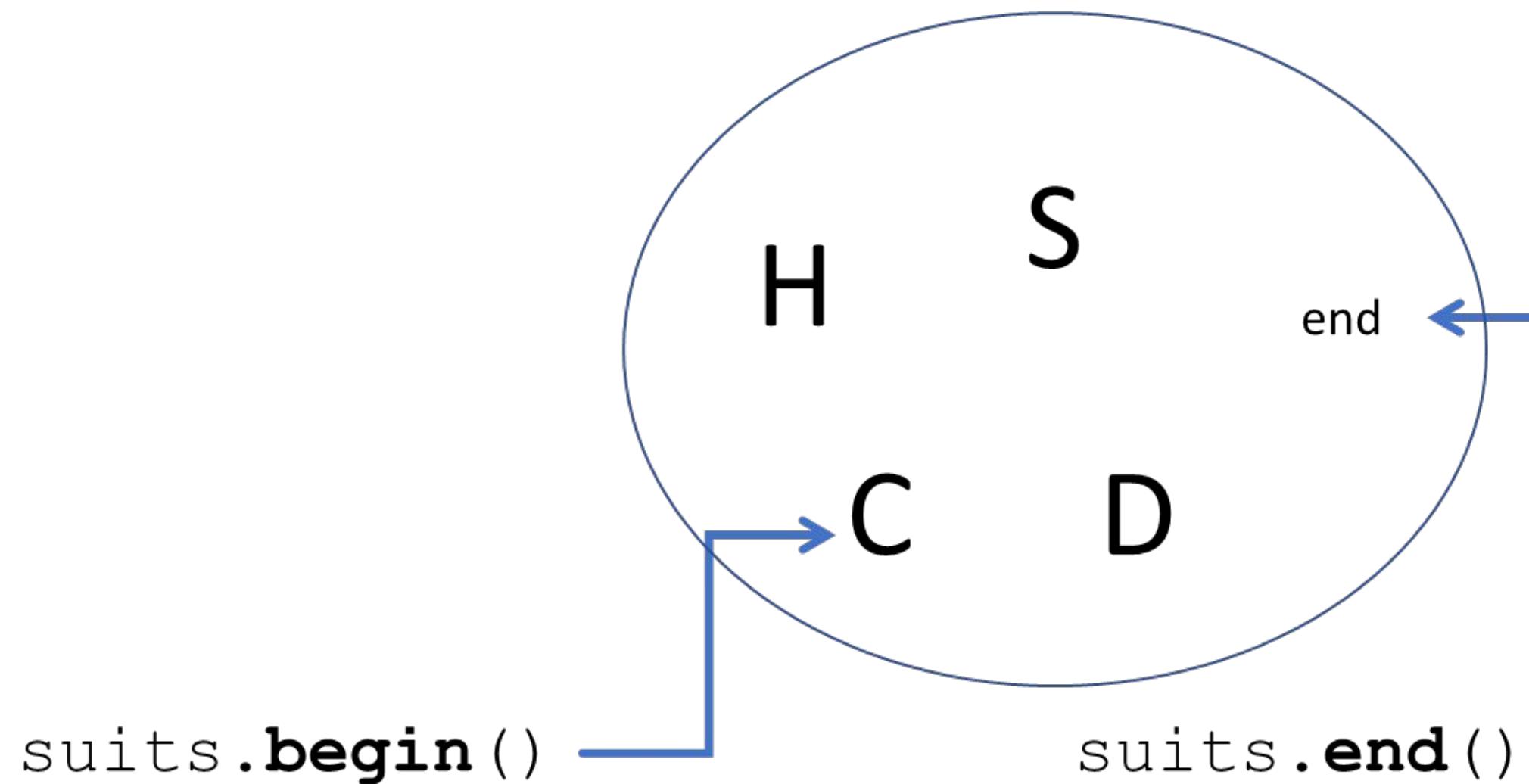
```
std::vector<int> vec {1, 2, 3};
```



The Standard Template Library

Declaring iterators

```
std::set<char> suits { 'C', 'H', 'S', 'D' };
```



The Standard Template Library

Initializing iterators

```
std::vector<int> vec {1,2,3};
```

```
std::vector<int>::iterator it = vec.begin();
```

or

```
auto it = vec.begin();
```

The Standard Template Library

Operations with iterators (it)

Operation	Description	Type of Iterator
<code>++it</code>	Pre-increment	All
<code>it++</code>	Post-increment	All
<code>it = it1</code>	Assignment	All
<code>*it</code>	Dereference	Input and Output
<code>it-></code>	Arrow operator	Input and Output
<code>it == it1</code>	Comparison for equality	Input
<code>it != it1</code>	Comparison for inequality	Input
<code>--it</code>	Pre-decrement	Bidirectional
<code>it--</code>	Post-decrement	Bidirectional
<code>it + i, it += i</code> <code>it - i, it -= i</code>	Increment and decrement	Random Access
<code>it < it1, it <= it1</code> <code>it > it1, it >= it1</code>	Comparison	Random Access

The Standard Template Library

Using iterators - std::vector

```
std::vector<int> vec {1,2,3};
```

```
std::vector<int>::iterator it = vec.begin();
```

```
while (it != vec.end()) {
    std::cout << *it << " ";
    ++it;
}
// 1 2 3
```

The Standard Template Library

Using iterators - std::vector

```
std::vector<int> vec {1,2,3};

for (auto it = vec.begin(); it != vec.end(); it++) {
    std::cout << *it << " ";
}

// 1 2 3
```

- This is how the range-based for loop works

The Standard Template Library

Using iterators - std::set

```
std::set<char> suits { 'C', 'H', 'S', 'D';

auto it = suits.begin();
while (it != suits.end()) {
    std::cout << *it << " " << std::end;
    ++it;
}
// C H S D
```

The Standard Template Library

Reverse iterators

- Works in reverse
- Last element is the first and first is the last
- `++` moves backward, `--` moves forward

```
std::vector<int> vec {1,2,3};  
std::vector<int>::reverse_iterator it = vec.begin();  
while (it != vec.end()) {  
    std::cout << *it << " ";  
    ++it;  
}  
// 3 2 1
```

The Standard Template Library

Other iterators

- **begin()** and **end()**
 - iterator
- **cbegin()** and **cend()**
 - const_iterator
- **rbegin()** and **rend()**
 - reverse_iterator
- **crbegin()** and **crend()**
 - const_reverse_iterator

The Standard Template Library

Algorithms

- STL algorithms work on sequences of container elements provided to them by an iterator
- STL has many common and useful algorithms
- Too many to describe in this section
 - <http://en.cppreference.com/w/cpp/algorith>
- Many algorithms require extra information in order to do their work
 - Functors (function objects)
 - Function pointers
 - Lambda expressions (C++11)

The Standard Template Library

Algorithms and iterators

- `#include <algorithm>`
- Different containers support different types of iterators
 - Determines the types of algorithms supported
- All STL algorithms expect iterators as arguments
 - Determines the sequence obtained from the container

The Standard Template Library

Iterator invalidation

- Iterators point to container elements
- It's possible iterators become invalid during processing
- Suppose we are iterating over a vector of 10 elements
 - And we clear() the vector while iterating? What happens?
 - Undefined behavior – our iterators are pointing to invalid locations

The Standard Template Library

Example algorithm - find with primitive types

- The `find` algorithm tries to locate the first occurrence of an element in a container
- Lots of variations
- Returns an iterator pointing to the located element or `end()`

```
std::vector<int> vec {1, 2, 3};
```

```
auto loc = std::find(vec.begin(), vec.end(), 3);
```

```
if (loc != vec.end()) // found it!
    std::cout << *loc << std::endl; // 3
```

The Standard Template Library

Example algorithm - find with user-defined types

- Find needs to be able to compare object
- operator== is used and must be provided by your class

```
std::vector<Player> team { /* assume initialized */ }
Player p {"Hero", 100, 12};

auto loc = std::find(team.begin(), team.end(), p);

if (loc != vec.end())                                // found it!
    std::cout << *loc << std::endl;                  // operator<< called
```

The Standard Template Library

Example algorithm - `for_each`

- `for_each` algorithm applies a function to each element in the iterator sequence
- Function must be provided to the algorithm as:
 - Functor (function object)
 - Function pointer
 - Lambda expression (C++11)
- Let's square each element

The Standard Template Library

for_each - using a functor

```
struct Square_Functor {  
    void operator()(int x) { // overload () operator  
        std::cout << x * x << " ";  
    }  
};  
Square_Functor square; // Function object  
  
std::vector<int> vec {1, 2, 3, 4};  
  
std::for_each(vec.begin(), vec.end(), square);  
// 1 4 9 16
```

The Standard Template Library

for_each - using a function pointer

```
void square(int x) { // function
    std::cout << x * x << " ";
}

std::vector<int> vec {1, 2, 3, 4};

std::for_each(vec.begin(), vec.end(), square);
// 1 4 9 16
```

The Standard Template Library

for_each - using a lambda expression

```
std::vector<int> vec {1, 2, 3, 4};
```

```
std::for_each(vec.begin(), vec.end(),
    [](int x) { std::cout << x * x << " "; }) // lambda
```

```
// 1 4 9 16
```

The Standard Template Library

std::array (C++11)

```
#include <array>
```

- Fixed size
 - Size must be known at compile time
- Direct element access
- Provides access to the underlying raw array
- Use instead of raw arrays when possible
- All iterators available and do not invalidate

The Standard Template Library

std::array - initialization and assignment

```
std::array<int, 5> arr1 { {1,2,3,4,5} } ; C++11 vs. C++14
```

```
std::array<std::string, 3> stooges {
    std::string("Larry"),
    "Moe",
    std::string("Curly")
};
```

```
arr1 = {2,4,6,8,10};
```

The Standard Template Library

std::array - common methods

```
std::array<int, 5> arr {1,2,3,4,5};
```

```
std::cout << arr.size();           // 5
```

```
std::cout << arr.at(0);          // 1
```

```
std::cout << arr[1];            // 2
```

```
std::cout << arr.front();        // 1
```

```
std::cout << arr.back();         // 5
```

The Standard Template Library

std::array - common methods

```
std::array<int, 5> arr {1,2,3,4,5};
```

```
std::array<int, 5> arr1 {10,20,30,40,50};
```

```
std::cout << arr.empty();           // 0 (false)
```

```
std::cout << arr.max_size();      // 5
```

```
arr.swap(arr1);                  // swaps the 2 arrays
```

```
int *data = arr.data();          // get raw array address
```

The Standard Template Library

std::vector

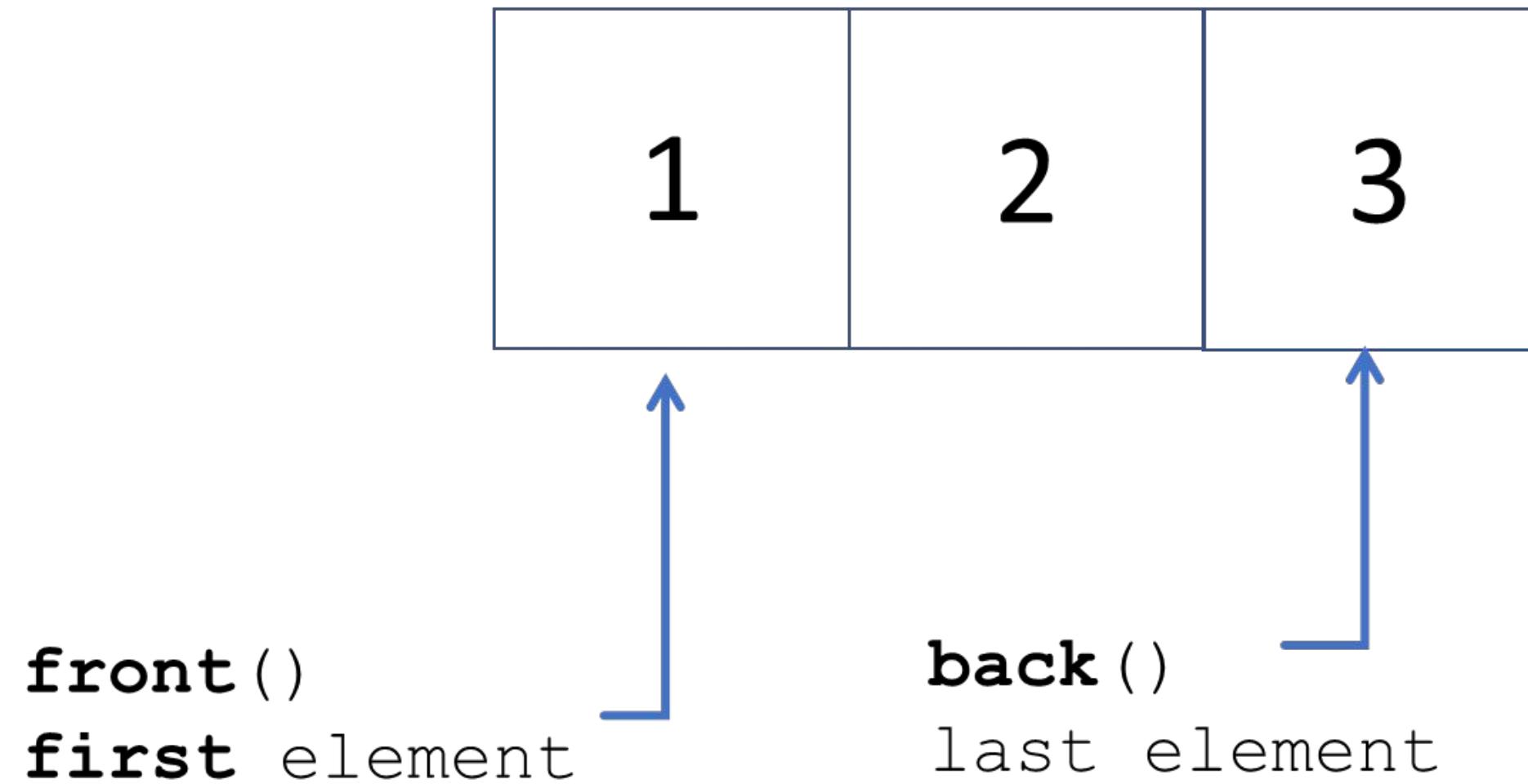
```
#include <vector>
```

- Dynamic size
 - Handled automatically
 - Can expand and contract as needed
 - Elements are stored in contiguous memory as an array
- Direct element access (constant time)
- Rapid insertion and deletion at the back (constant time)
- Insertion or removal of elements (linear time)
- All iterators available and may invalidate

The Standard Template Library

std::vector

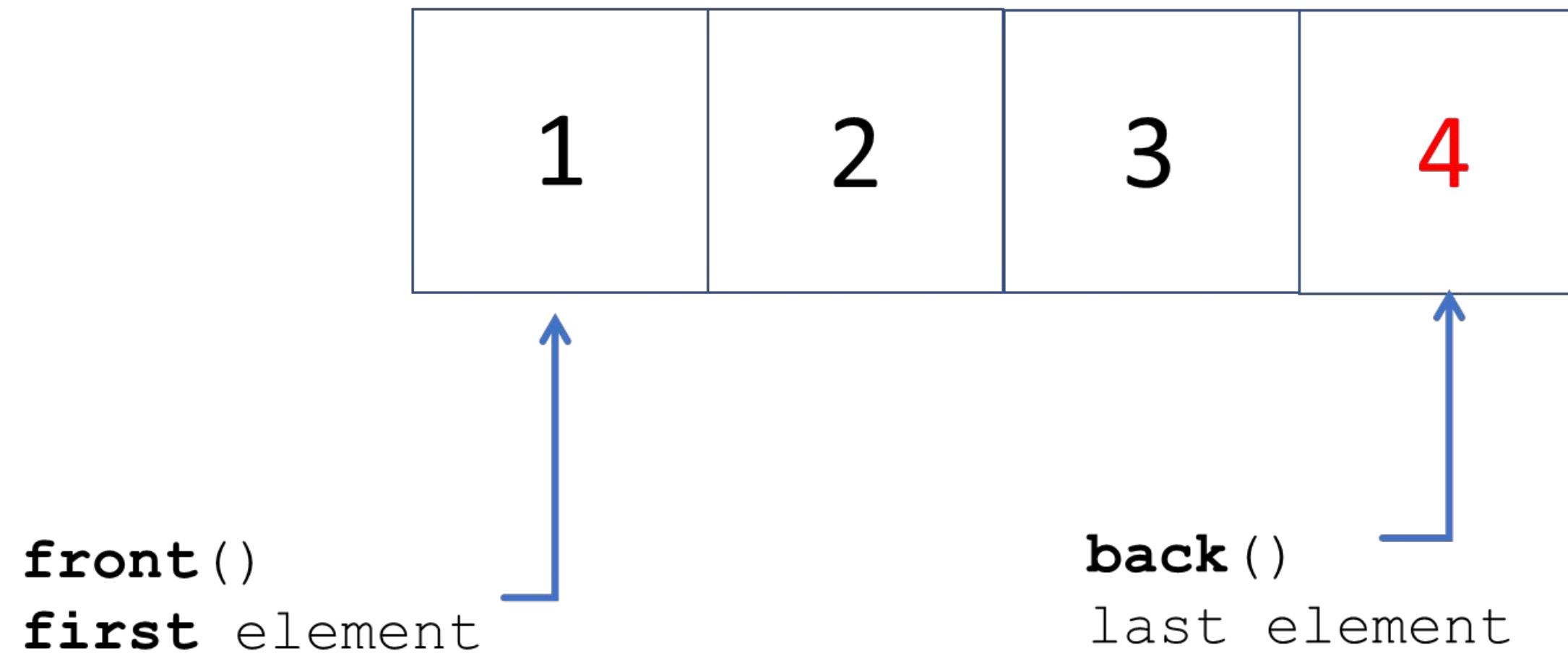
```
std::vector<int> vec {1, 2, 3};
```



The Standard Template Library

std::vector

vec.**push_back**(**4**)



The Standard Template Library

std::vector - initialization and assignment

```
std::vector<int> vec {1,2,3,4,5};  
std::vector<int> vec1 (10,100); // ten 100s
```

```
std::vector<std::string> stooges {  
    std::string{"Larry"},  
    "Moe",  
    std::string{"Curly"}  
};
```

```
vec1 = {2,4,6,8,10};
```

The Standard Template Library

std::vector - common methods

```
std::vector<int> vec {1,2,3,4,5};
```

```
std::cout << vec.size(); // 5
```

```
std::cout << vec.capacity(); // 5
```

```
std::cout << vec.max_size(); // a very large number
```

```
std::cout << vec.at(0); // 1
```

```
std::cout << vec[1]; // 2
```

```
std::cout << vec.front(); // 1
```

```
std::cout << vec.back(); // 5
```

The Standard Template Library

std::vector - common methods

```
Person p1 {"Larry", 18};
```

```
std::vector<Person> vec;
```

```
vec.push_back(p1);           // add p1 to the back
```

```
vec.pop_back();              // remove p1 from the back
```

```
vec.push_back(Person{"Larry", 18});
```

```
vec.emplace_back("Larry", 18);      // efficient!!
```

The Standard Template Library

std::vector - common methods

```
std::vector<int> vec1 {1,2,3,4,5};
```

```
std::vector<int> vec2 {10,20,30,40,50};
```

```
std::cout << vec1.empty();           // 0 (false)
```

```
vec1.swap(vec2);                  // swaps the 2 vector
```

```
std::sort(vec1.begin(), vec1.end());
```

The Standard Template Library

std::vector - common methods

```
std::vector<int> vec1 {1,2,3,4,5};
```

```
std::vector<int> vec2 {10,20,30,40,50};
```

```
auto it = std::find(vec1.begin(), vec1.end(), 3);
```

```
vec1.insert(it, 10); // 1,2,10,3,4,5
```

```
it = std::find(vec1.begin(), vec1.end(), 4);
```

```
vec1.insert(it, vec2.begin(), vec2.end());
```

```
// 1,2,10,3,10,20,30,40,50,4,5
```

The Standard Template Library

std::deque (double ended queue)

```
#include <deque>
```

- Dynamic size
 - Handled automatically
 - Can expand and contract as needed
 - Elements are NOT stored in contiguous memory
- Direct element access (constant time)
- Rapid insertion and deletion at the front **and** back (constant time)
- Insertion or removal of elements (linear time)
- All iterators available and may invalidate

The Standard Template Library

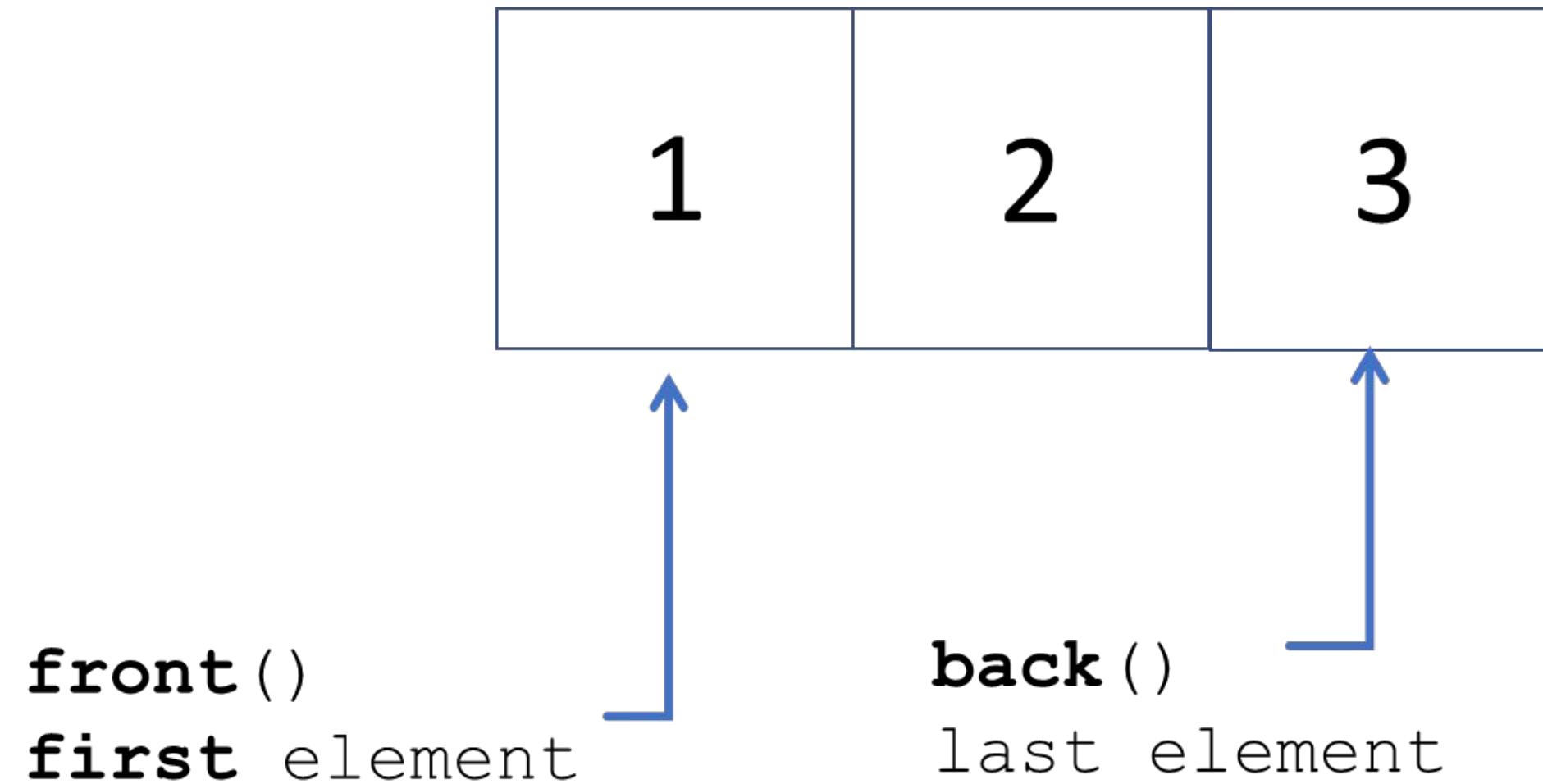
std::deque - initialization and assignment

```
std::deque<int> d{1,2,3,4,5};  
std::deque<int> d1(10,100);           // ten 100s  
  
std::deque<std::string> stooges {  
    std::string("Larry"),  
    "Moe",  
    std::string("Curly")  
};  
  
d = {2,4,6,8,10};
```

The Standard Template Library

std::deque

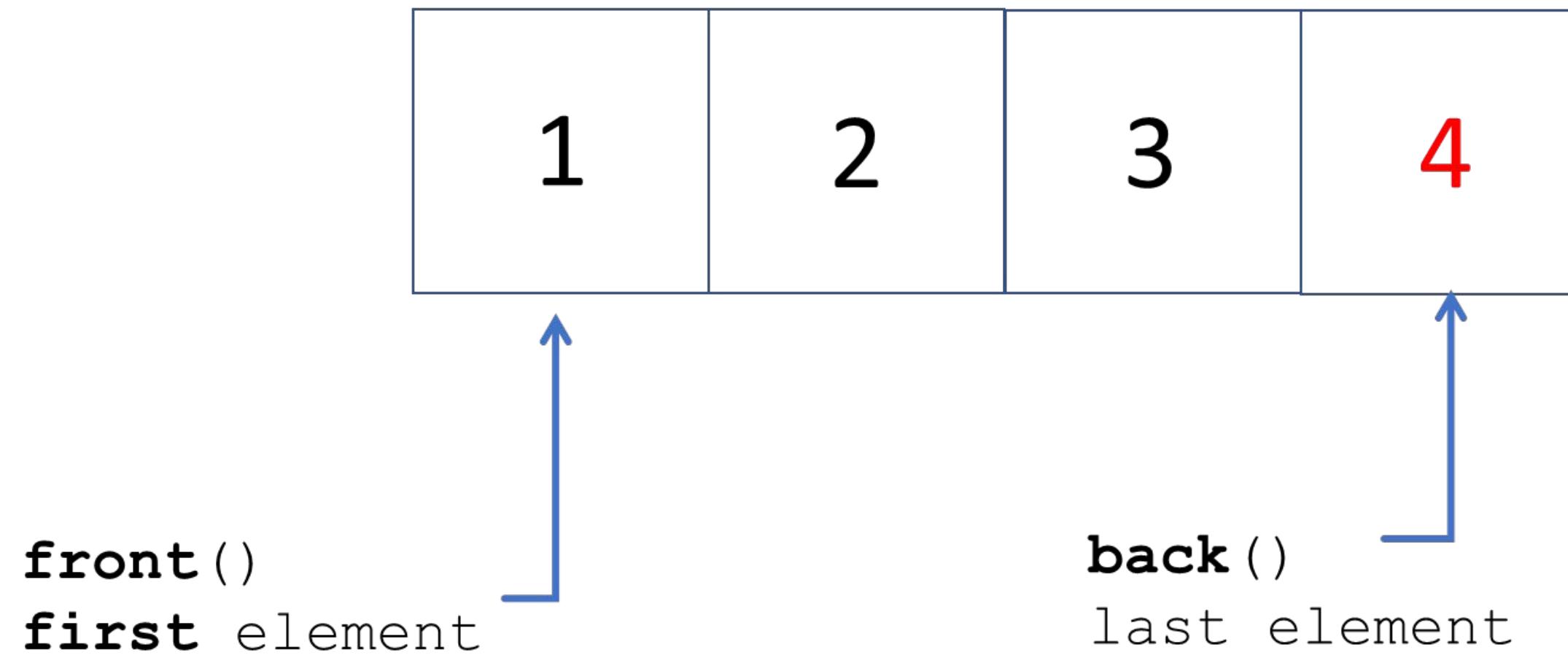
```
std::deque<int> d{1, 2, 3};
```



The Standard Template Library

std::deque

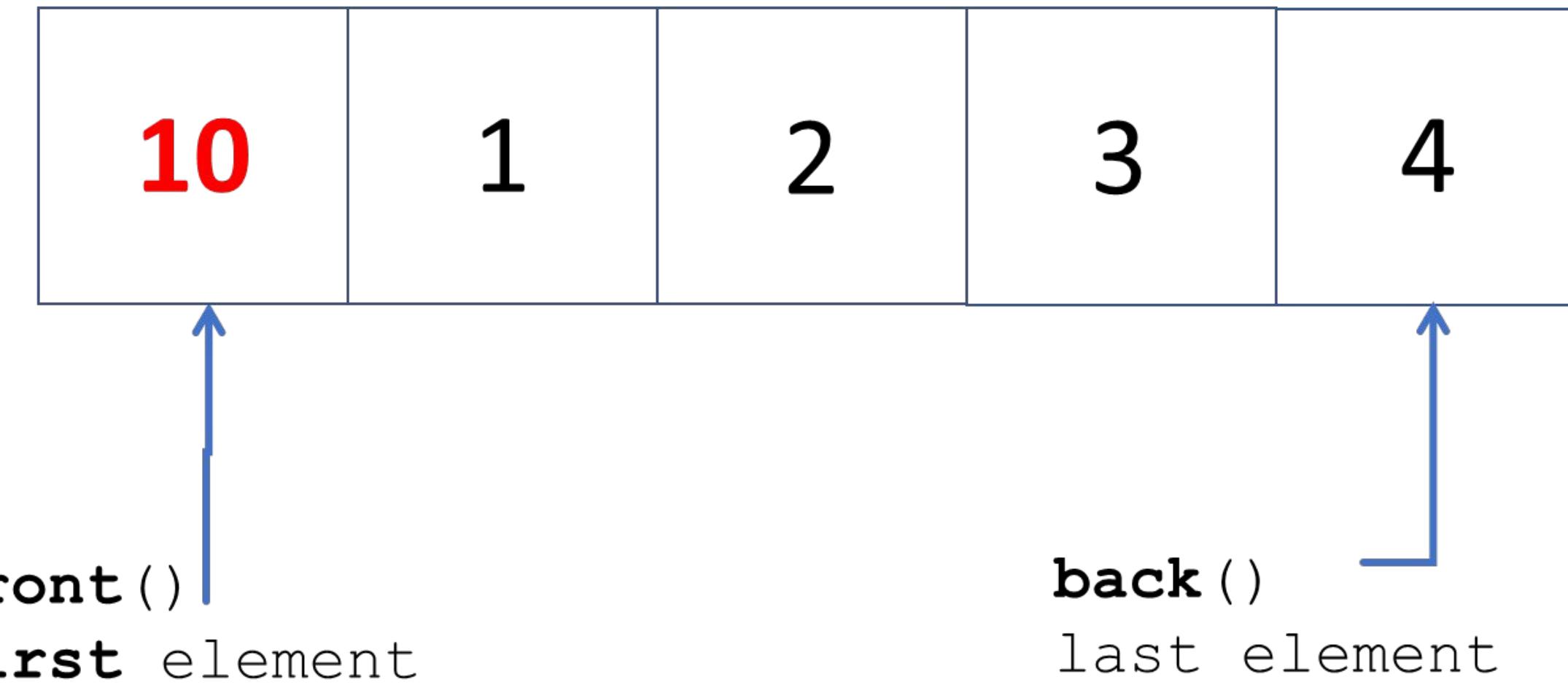
d.**push_back**(**4**)



The Standard Template Library

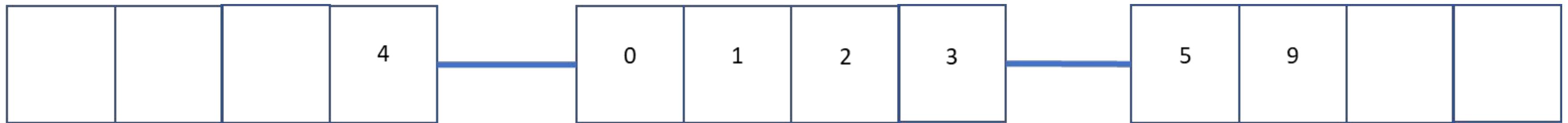
std::deque

d.**push_front(10)**



The Standard Template Library

std::deque



The Standard Template Library

std::deque - common methods

```
std::deque<int> d {1,2,3,4,5};
```

```
std::cout << d.size(); // 5
```

```
std::cout << d.max_size(); // a very large number
```

```
std::cout << d.at(0); // 1
```

```
std::cout << d[1]; // 2
```

```
std::cout << d.front(); // 1
```

```
std::cout << d.back(); // 5
```

The Standard Template Library

std::deque - common methods

```
Person p1 {"Larry", 18};
```

```
std::deque<Person> d;
```

```
d.push_back(p1);           // add p1 to the back
```

```
d.pop_back();              // remove p1 from the back
```

```
d.push_front(Person{"Larry", 18});
```

```
d.pop_front();             // remove element from the front
```

```
d.emplace_back("Larry", 18); // add to back efficient!
```

```
d.emplace_front("Moe", 24); // add to front
```

The Standard Template Library

`std::list` and `std::forward_list`

- Sequence containers
- Non-contiguous in memory
- No direct access to elements
- Very efficient for inserting and deleting elements once an element is found

The Standard Template Library

std::list

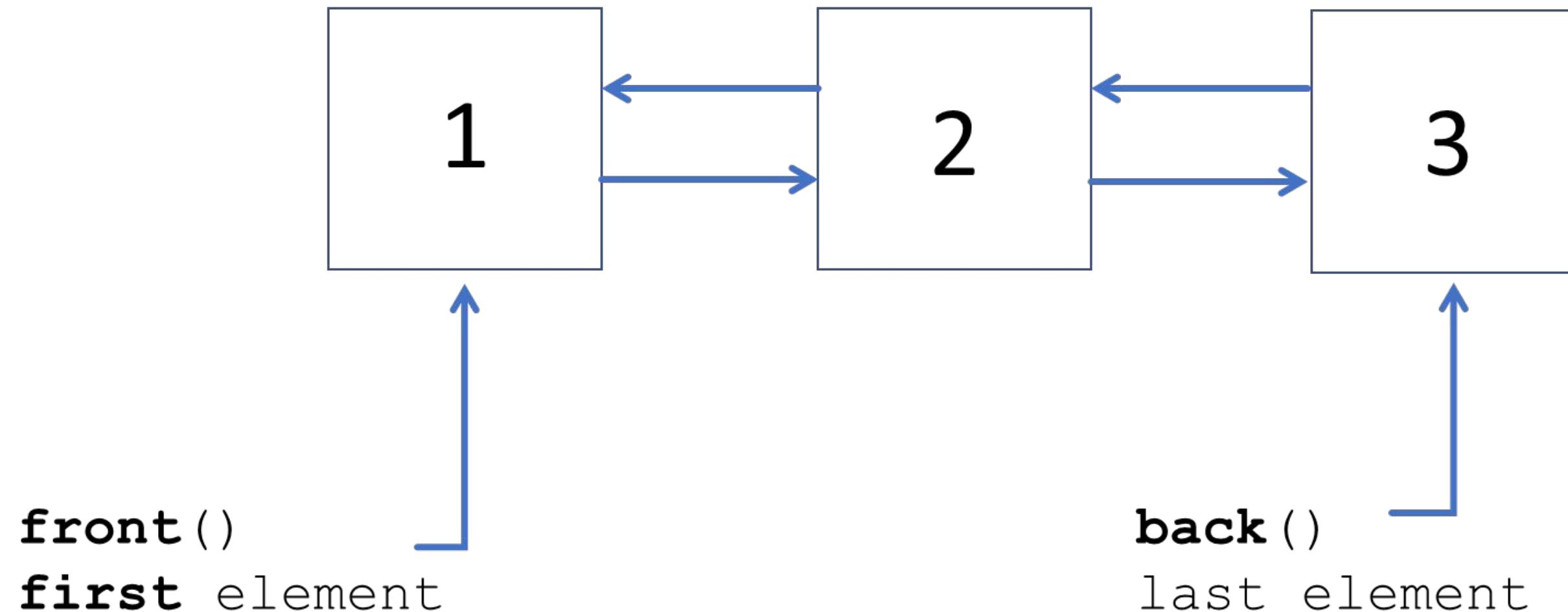
```
#include <list>
```

- Dynamic size
 - Lists of elements
 - list is bidirectional (doubly-linked)
- Direct element access is NOT provided
- Rapid insertion and deletion of elements anywhere in the container (constant time)
- All iterators available and invalidate when corresponding element is deleted

The Standard Template Library

std::list

```
std::list<int> l{1, 2, 3};
```



The Standard Template Library

std::list - initialization and assignment

```
std::list<int> l {1,2,3,4,5};  
std::list<int> ll(10,100); // ten 100s
```

```
std::list<std::string> stooges {  
    std::string{"Larry"},  
    "Moe",  
    std::string{"Curly"}  
};
```

```
l = {2,4,6,8,10};
```

The Standard Template Library

std::list - common methods

```
std::list<int> l {1,2,3,4,5};
```

```
std::cout << l.size();           // 5
```

```
std::cout << l.max_size();      // a very large number
```

```
std::cout << l.front();         // 1
```

```
std::cout << l.back();          // 5
```

The Standard Template Library

std::list - common methods

```
Person p1 {"Larry", 18};  
std::list<Person> l;
```

```
l.push_back(p1);           // add p1 to the back  
l.pop_back();             // remove p1 from the back  
  
l.push_front(Person{"Larry", 18});  
l.pop_front();            // remove element from the front  
  
l.emplace_back("Larry", 18); // add to back efficient!!  
l.emplace_front("Moe", 24); // add to front
```

The Standard Template Library

std::list - methods that use iterators

```
std::list<int> l {1,2,3,4,5};  
auto it = std::find(l.begin(), l.end(), 3);  
  
l.insert(it, 10);           // 1 2 10 3 4 5  
  
l.erase(it);               // erases the 3,   1 2 10 4 5  
  
l.resize(2);               // 1 2  
  
l.resize(5);               // 1 2 0 0 0
```

The Standard Template Library

std::list - common methods

```
// traversing the list (bi-directional)

std::list<int> l {1,2,3,4,5};
auto it = std::find(l.begin(), l.end(), 3);

std::cout << *it;                                // 3
it++;
std::cout << *it;                                // 4
it--;
std::cout << *it;                                // 3
```

The Standard Template Library

`std::forward_list`

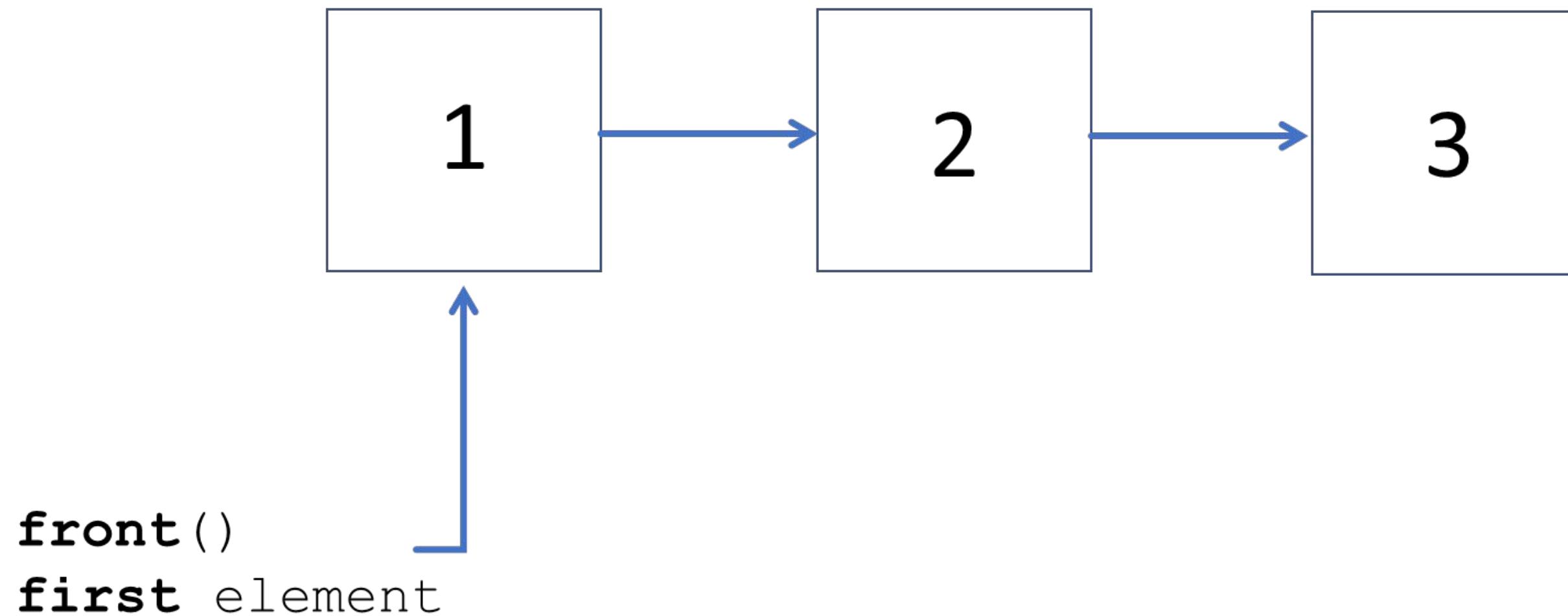
```
#include <forward_list>
```

- Dynamic size
 - Lists of elements
 - list uni-directional (singly-linked)
 - Less overhead than a `std::list`
- Direct element access is NOT provided
- Rapid insertion and deletion of elements anywhere in the container (constant time)
- Reverse iterators not available. Iterators invalidate when corresponding element is deleted

The Standard Template Library

`std::forward_list`

```
std::forward_list<int> l{1,2,3};
```



The Standard Template Library

std::forward_list - common methods

```
std::forward_list<int> l {1,2,3,4,5};
```

```
std::cout << l.size(); // Not available
```

```
std::cout << l.max_size(); // a very large number
```

```
std::cout << l.front(); // 1
```

```
std::cout << l.back(); // Not available
```

The Standard Template Library

std::forward_list - common methods

```
Person p1 {"Larry", 18};
```

```
std::forward_list<Person> l;
```

```
l.push_front(p1);           // add p1 to the front
```

```
l.pop_front();              // remove p1 from the front
```

```
l.emplace_front("Moe", 24); // add to front
```

The Standard Template Library

`std::forward_list` - methods that use iterators

```
std::forward_list<int> l {1,2,3,4,5};  
auto it = std::find(l.begin(), l.end(), 3);  
  
l.insert_after(it, 10);      // 1 2 3 10 4 5  
l.emplace_after(it, 100);   // 1 2 3 100 10 4 5  
  
l.erase_after(it);         // erases the 100, 1 2 3 10 4 5  
  
l.resize(2);               // 1 2  
  
l.resize(5);               // 1 2 0 0 0
```

The Standard Template Library

The STL Set containers

- Associative containers

- Collection of stored objects that allow fast retrieval using a key
- STL provides Sets and Maps
- Usually implemented as a balanced binary tree or hashsets
- Most operations are very efficient

- Sets

- `std::set`
- `std::unordered_set`
- `std::multiset`
- `std::unordered_multiset`

The Standard Template Library

std::set

```
#include <set>
```

- Similar to a mathematical set
- Ordered by key
- No duplicate elements
- All iterators available and invalidate when corresponding element is deleted

The Standard Template Library

std::set - initialization and assignment

```
std::set<int> s {1,2,3,4,5};
```

```
std::set<std::string> stooges {
    std::string{"Larry"},
    "Moe",
    std::string{"Curly"}
};
```

```
s = {2,4,6,8,10};
```

The Standard Template Library

std::set - common methods

```
std::set<int> s {4,1,1,3,3,2,5};      // 1 2 3 4 5
```

```
std::cout << s.size();                // 5
```

```
std::cout << s.max_size();           // a very large number
```

- No concept of front and back

```
s.insert(7);                      // 1 2 3 4 5 7
```

The Standard Template Library

std::set - common methods

```
Person p1 {"Larry", 18};
```

```
Person p2 {"Moe", 25};
```

```
std::set<Person> stooges;
```

```
stooges.insert(p1);           // adds p1 to the set
```

```
auto result = stooges.insert(p2); // adds p2 to the set
```

- **uses operator< for ordering!**
- **returns a std::pair<iterator, bool>**
 - first is an iterator to the inserted element or to the duplicate in the set
 - second is a boolean indicating success or failure

The Standard Template Library

std::set - common methods

```
std::set<int> s {1,2,3,4,5};  
  
s.erase(3); // erase the 3 : 1 2 4 5
```

```
auto it = s.find(5);  
if (it != s.end())  
    s.erase(it); // erase the 5: 1 2 4
```

The Standard Template Library

std::set - common methods

```
std::set<int> s {1,2,3,4,5};
```

```
int num = s.count(1); // 0 or 1
```

```
s.clear(); // remove all elements
```

```
s.empty(); // true or false
```

The Standard Template Library

std::multi_set

```
#include <set>
```

- Sorted by key
- Allows duplicate elements
- All iterators are available

The Standard Template Library

std::unordered_set

```
#include <unordered_set>
```

- Elements are unordered
- No duplicate elements allowed
- Elements cannot be modified
 - Must be erased and new element inserted
- No reverse iterators are allowed

The Standard Template Library

std::unordered_multiset

```
#include <unordered_set>
```

- Elements are unordered
- Allows duplicate elements
- No reverse iterators are allowed

The Standard Template Library

The STL Map containers

- Associative containers

- Collection of stored objects that allow fast retrieval using a key
- STL provides Sets and Maps
- Usually implemented as a balanced binary tree or hashsets
- Most operations are very efficient

- Maps

- `std::map`
- `std::unordered_map`
- `std::multimap`
- `std::unordered_multimap`

The Standard Template Library

std::map

```
#include <map>
```

- Similar to a dictionary
- Elements are stored as Key, Value pairs (std::pair)
- Ordered by key
- No duplicate elements (keys are unique)
- Direct element access using the key
- All iterators available and invalidate when corresponding element is deleted

The Standard Template Library

std::map - initialization and assignment

```
std::map<std::string, int> m1 {  
    {"Larry", 18},  
    {"Moe", 25}  
};
```

```
std::map<std::string, std::string> m2 {  
    {"Bob", "Butcher"},  
    {"Anne", "Baker"},  
    {"George", "Candlestick maker"}  
};
```

The Standard Template Library

std::map - common methods

```
std::map<std::string, std::string> m {  
    {"Bob", "Butcher"},  
    {"Anne", "Baker"},  
    {"George", "Candlestick maker"}  
};
```

```
std::cout << m.size();                                // 3  
std::cout << m.max_size();                            // a very large number
```

- No concept of front and back

The Standard Template Library

std::map - common methods

```
std::map<std::string, std::string> m {  
    {"Bob", "Butcher"},  
    {"Anne", "Baker"},  
    {"George", "Candlestick maker"}  
};
```

```
std::pair<std::string, std::string> p1 {"James", "Mechanic"};  
  
m.insert(p1);  
  
m.insert(std::make_pair("Roger", "Ranger"));
```

The Standard Template Library

std::map - common methods

```
std::map<std::string, std::string> m {  
    {"Bob", "Butcher"},  
    {"Anne", "Baker"},  
    {"George", "Candlestick maker"}  
};  
  
m["Frank"] = "Teacher";           // insert  
  
m["Frank"] = "Instructor";        // update value  
m.at("Frank") = "Professor";     // update value
```

The Standard Template Library

std::map - common methods

```
std::map<std::string, std::string> m {  
    {"Bob", "Butcher"},  
    {"Anne", "Baker"},  
    {"George", "Candlestick maker"}  
};  
  
m.erase("Anne");                                // erase Anne  
  
if (m.find("Bob") != m.end())                   // find Bob  
    std::cout << "Found Bob!";  
  
auto it = m.find("George");  
if (it != m.end())  
    m.erase(it);                                 // erase George
```

The Standard Template Library

std::map - common methods

```
std::map<std::string, std::string> m {  
    {"Bob", "Butcher"},  
    {"Anne", "Baker"},  
    {"George", "Candlestick maker"}  
};  
  
int num = m.count("Bob"); // 0 or 1  
  
m.clear(); // remove all elements  
  
m.empty(); // true or false
```

The Standard Template Library

std::multi_map

```
#include <map>
```

- Ordered by key
- Allows duplicate elements
- All iterators are available

The Standard Template Library

std::unordered_map

```
#include <unordered_map>
```

- Elements are unordered
- No duplicate elements allowed
- No reverse iterators are allowed

The Standard Template Library

std::unordered_multimap

```
#include <unordered_map>
```

- Elements are unordered
- Allows duplicate elements
- No reverse iterators are allowed

The Standard Template Library

`std::stack`

- Last-in First-out (LIFO) data structure
- Implemented as an adapter over other STL container
Can be implemented as a vector, list, or deque
- All operations occur on one end of the stack (top)
- No iterators are supported

The Standard Template Library

std::stack

```
#include <stack>
```

- push – insert an element at the top of the stack
- pop – remove an element from the top of the stack
- top – access the top element of the stack
- empty – is the stack empty?
- size – number of elements in the stack

The Standard Template Library

std::stack - initialization

```
std::stack<int> s;                                // deque
```

```
std::stack<int, std::vector<int>> s1;      // vector
```

```
std::stack<int, std::list<int>> s2;      // list
```

```
std::stack<int, std::deque<int>> s3;      // deque
```

The Standard Template Library

std::stack - common methods

```
std::stack<int> s;
```

```
s.push(10);
```



```
s.push(20);
```



```
s.push(30);
```



The Standard Template Library

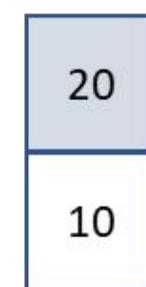
std::stack - common methods

```
std::cout << s.top();
```



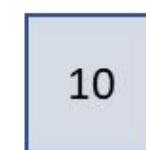
// 30

```
s.pop();
```



// 30 is removed

```
s.pop();
```



// 20 is removed

```
std::cout << s.size();
```

// 1

The Standard Template Library

`std::queue`

- First-in First-out (FIFO) data structure
- Implemented as an adapter over other STL container
Can be implemented as a list or deque
- Elements are pushed at the back and popped from the front
- No iterators are supported

The Standard Template Library

std::queue

```
#include <queue>
```

- push - insert an element at the back of the queue
- pop - remove an element from the front of the queue
- front - access the element at the front
- back - access the element at the back
- empty - is the queue empty?
- size - number of elements in the queue

The Standard Template Library

std::queue - initialization

```
std::queue<int> q; // deque
```

```
std::queue<int, std::list<int>> q2; // list
```

```
std::queue<int, std::deque<int>> q3; // deque
```

The Standard Template Library

std::queue - common methods

```
std::queue<int> q;
```

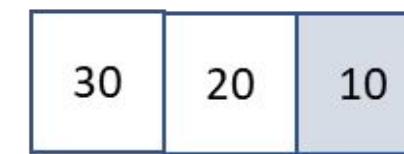
```
q.push(10);
```



```
q.push(20);
```



```
q.push(30);
```



The Standard Template Library

std::queue - common methods

```
std::cout << q.front();
```



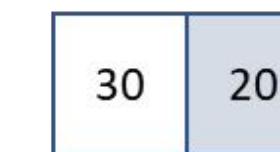
// 10

```
std::cout << q.back();
```



// 30

```
q.pop();
```



// remove 10

```
q.pop();
```



// remove 20

```
std::cout << q.size();
```

// 1

The Standard Template Library

`std::priority_queue`

- Allows insertions and removal of elements in order from the front of the container
- Elements are stored internally as a vector by default
- Elements are inserted in ***priority*** order
(largest value will always be at the front)
- No iterators are supported

The Standard Template Library

std::priority_queue

```
#include <queue>
```

- push - insert an element into sorted order
- pop - removes the top element (greatest)
- top - access the top element (greatest)
- empty - is the queue empty?
- size - number of elements in the queue

The Standard Template Library

std::priority_queue - initialization

```
std::priority_queue<int> pq; // vector
```

```
    pq.push(10);
```

```
    pq.push(20);
```

```
    pq.push(3);
```

```
    pq.push(4);
```

```
    std::cout << pq.top(); // 20 (largest)
```

```
    pq.pop(); // remove 20
```

```
    pq.top(); // 10 (largest)
```

Bonus Section Overview

C++ Lambda expressions

- What is a lambda expression?
 - Motivation
 - Review of function objects (functors)
 - Relation between lambdas and function objects
- Structure of a lambda expression
- Types of lambda expressions
 - Stateless lambda expression
 - Stateful lambda expression (capturing context)
- Lambdas and the STL

C++ Lambda Expressions

Motivation

- Prior to C++ 11
 - Function objects
 - Function pointers
- We often write many short functions that control algorithms
- These short functions would be encapsulated in small classes to produce function objects
- Many times the classes and functions are far removed from where they are used leading to modification, maintenance, and testing issues
- Compiler cannot effectively inline these functions for efficiency.

Motivation

Function objects

```
class Multiplier {  
private:  
    int num{};  
public:  
    Multiplier(int n) : num {n} {}  
    int operator()(int n) const {  
        return num * n;  
    }  
};
```

Motivation

Function objects

```
std::vector<int>    vec {1,2,3,4};  
Multiplier mult{10};  
  
std::transform(vec.begin(), vec.end(), vec.begin(),  
              mult);  
  
vec now contains {10,20,30,40}
```

Motivation

Function objects

```
std::vector<int>    vec {1,2,3,4};  
  
std::transform(vec.begin(), vec.end(), vec.begin(),  
              Multiplier(10));  
  
vec now contains {10,20,30,40}
```

Motivation

Generic function objects

```
template <typename T>
struct Displayer {
    void operator() (const T &data) {
        std::cout << data << " ";
    }
};
```

Motivation

Generic function objects

```
Displayer<int> d1;  
Displayer<std::string> d2;  
  
d1(100);           // d.operator(100);  
                    // displays 100  
d2("Frank");     // d.operator("Frank");  
                    // displays Frank
```

Motivation

Generic function objects

```
std::vector<int> vec1 {1,2,3,4,5};  
std::vector<std::string> vec2 {"Larry", "Moe", "Curly"};  
  
std::for_each(vec1.begin(), vec1.end(), Displayer<int>());  
  
std::for_each(vec1.begin(), vec1.end(), d1);  
  
std::for_each(vec2.begin(), vec2.end(), d2);
```

Motivation

Using a lambda expression

```
std::vector<int> vec1 {1,2,3,4,5};  
std::vector<std::string> vec2 {"Larry", "Moe", "Curly"};  
  
std::for_each(vec1.begin(), vec1.end(),  
             [](int x) { std::cout << x << " "; } );  
  
std::for_each(vec2.begin(), vec2.end(),  
             [](std::string s) { std::cout << s << " "; } );
```

C++ Lambda Expressions

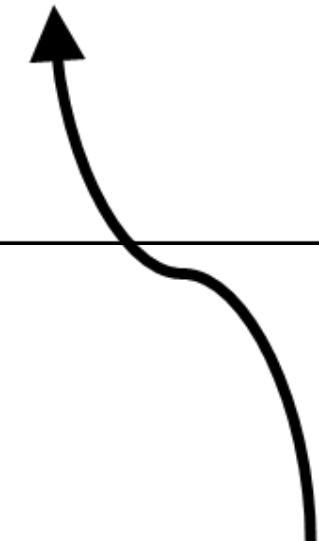
The Structure of a Lambda Expression

```
[ ] () -> return_type specifiers { };
```

C++ Lambda Expressions

The Structure of a Lambda Expression

```
[ ] () -> return_type specifiers { };
```



*Capture List
Defines the start of the lambda*

C++ Lambda Expressions

The Structure of a Lambda Expression

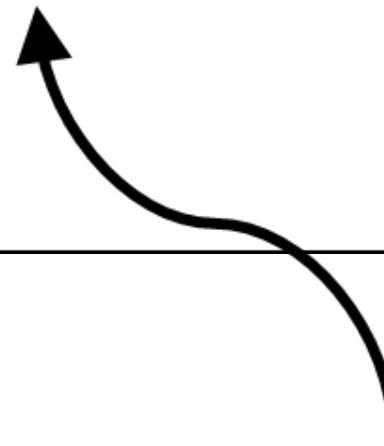
```
[ ]  () -> return_type specifiers { } ;
```

Parameter List
Comma separated list of parameters

C++ Lambda Expressions

The Structure of a Lambda Expression

```
[ ] () -> return_type specifiers { } ;
```



Return type
Can be omitted and the compiler
will try to deduce it!

C++ Lambda Expressions

The Structure of a Lambda Expression

```
[ ] () -> return_type specifiers { };
```

Optional specifiers
mutable and constexpr

C++ Lambda Expressions

The Structure of a Lambda Expression

```
[ ] () -> return_type specifiers { } ;
```

*Body
Your code!*

C++ Lambda Expressions

A simple lambda expression

```
[ ]  ()  { std::cout << "Hi"; } ;
```

C++ Lambda Expressions

```
[ ] () { std::cout << "Hi"; }();  
// Displays Hi
```

C++ Lambda Expressions

Parameters to lambda expressions

```
[ ] (int x) { std::cout << x; };
```

C++ Lambda Expressions

Parameters to lambda expressions

```
[ ] (int x, int y) { std::cout << x + y; };
```

C++ Lambda Expressions

Assigning a lambda expression to a variable

```
auto l = [] () { std::cout << "Hi" ; } ;  
  
l () ; // Displays Hi
```

C++ Lambda Expressions

Assigning a lambda expression to a variable

```
auto l = [] (int x) { std::cout << x; };

l(10); // displays 10
l(100); // displays 100
```

C++ Lambda Expressions

Returning a value from a lambda expression

```
auto l = [] (int x, int y) -> int { return x + y; };
```

or

```
auto l = [] (int x, int y) { return x + y; };
```

```
std::cout << l(2,3); // displays 5
```

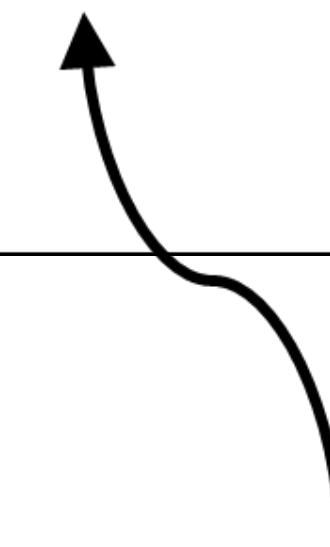
```
std::cout << l(10,20); // display 30
```

C++ Stateless Lambda Expressions

```
[ ] () -> return_type specifiers { };
```

C++ Stateless Lambda Expressions

```
[ ] () -> return_type specifiers { };
```



*Capture List
Defines the context in which the lambda executes*

C++ Stateless Lambda Expressions

Simple stateless lambda expressions

```
[ ] () { std::cout << "Hi"; } () ;           // Displays Hi

int x {10};

[ ] (int x) { std::cout << x; } (100);
```

C++ Stateless Lambda Expressions

Simple stateless lambda expressions

```
[ ] () { std::cout << "Hi"; } () ;           // Displays Hi

int x {10};

[ ] (int x) { std::cout << x; } (100);      // Displays 100
```

C++ Stateless Lambda Expressions

Simple stateless lambda expressions

```
const int n {3};  
int nums[n] {10,20,30};  
  
auto sum = [] (int nums[], int n) {  
    int sum {0};  
    for (int i = 0; i < n; i++)  
        sum += nums[i];  
    return sum;  
};  
  
std::cout << sum(nums, 3); // Displays 60
```

C++ Stateless Lambda Expressions

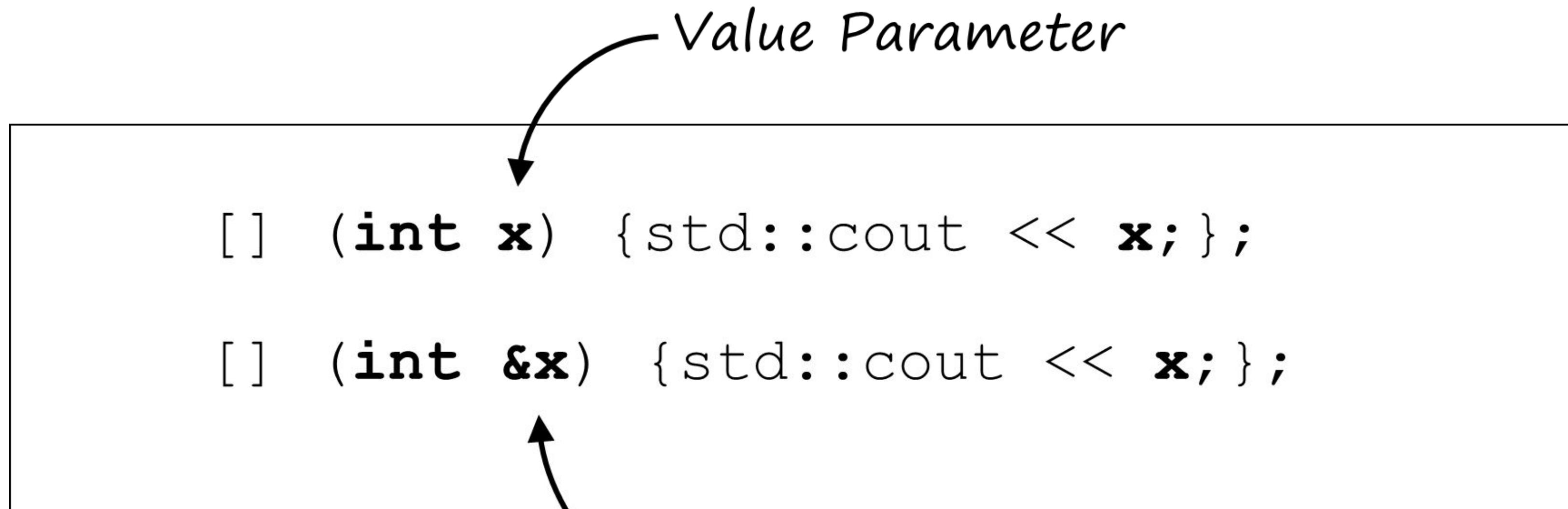
Using values and references as lambda parameters

Value Parameter

```
[ ] (int x) { std::cout << x; };
```

C++ Stateless Lambda Expressions

Using values and references as lambda parameters



C++ Stateless Lambda Expressions

Using values and references as lambda parameters

```
int test_score1 {88};  
int test_score2 {75};  
  
auto bonus = [] (int &score1, int &score2, int bonus_points) {  
    score1 += bonus_points;  
    score2 += bonus_points;  
};  
  
bonus(test_score1, test_score2, 5);  
  
std::cout << "test_score1: " << test_score1 << std::endl;      // Displays 93  
std::cout << "test_score2: " << test_score2 << std::endl;      // Displays 80
```

C++ Stateless Lambda Expressions

Using pointers as lambda parameters

Pointer Parameter

```
int x;  
  
auto l = [] (int *x) { std::cout << *x; } ;  
  
l (&x);
```

C++ Stateless Lambda Expressions

Using pointers as lambda parameters

Pointer Parameter

```
int x;  
  
auto l = [] (int *x) { std::cout << *x; };  
  
l (&x);
```



Referencing (address-of) Operator

C++ Stateless Lambda Expressions

Using pointers as lambda parameters

Pointer Parameter

```
int x;  
  
auto l = [] (int *x) { std::cout << *x; };  
  
l (&x);
```

Dereferencing Operator

Referencing (address-of) Operator

C++ Stateless Lambda Expressions

Using pointers as lambda parameters

```
int test_score1 {88};  
int test_score2 {75};  
  
auto bonus = [] (int *score1, int *score2, int bonus_points) {  
    *score1 += bonus_points;  
    *score2 += bonus_points;  
};  
  
bonus(&test_score1, &test_score2, 5);  
  
std::cout << "test_score1: " << test_score1 << std::endl;      // Displays 93  
std::cout << "test_score2: " << test_score2 << std::endl;      // Displays 80
```

C++ Stateless Lambda Expressions

Using arrays and vectors as lambda reference parameters

```
std::vector<int> test_scores {93,88,75,68,65};

auto bonus = [] (std::vector<int> &scores, int bonus_points) {
    for (int &score: scores)
        score += bonus_points;
};

bonus(test_scores, 5);

std::cout << "test_scores:" << std::endl;
std::cout << test_scores[0] << std::endl;           // Displays 98
std::cout << test_scores[1] << std::endl;           // Displays 93
std::cout << test_scores[2] << std::endl;           // Displays 80
std::cout << test_scores[3] << std::endl;           // Displays 73
std::cout << test_scores[4] << std::endl;           // Displays 70
```

C++ Stateless Lambda Expressions

Using auto as lambda parameter type specifiers

```
int num1 {10};  
float num2 {20.5};  
  
auto l = [] ( x) {std::cout << x;};  
  
l(num1);  
l(num2);
```

C++ Stateless Lambda Expressions

Using auto as lambda parameter type specifiers

```
int num1 {10};  
float num2 {20.5};  
  
auto l = [] (auto x) {std::cout << x;};  
  
l(num1);  
l(num2);
```

C++ Stateless Lambda Expressions

Using auto as lambda parameter type specifiers

```
std::vector<int> test_scores1 {93,88,75,68,65};  
std::vector<float> test_scores2 {88.5,85.5,75.5,68.5,65.5};  
  
auto bonus = [] (auto &scores, int bonus_points) {  
    for (auto &score: scores)  
        score += bonus_points;  
};  
  
bonus(test_scores1, 5);           // Valid  
bonus(test_scores2, 5);           // Valid
```

C++ Stateless Lambda Expressions

Using lambda expressions as function parameters

```
#include <functional> // for std::function

void foo(std::function<void(int)> l) { // C++14
    l(10);
}

or

void foo(void (*l)(int)) { // C++14
    l(10);
}

or

void foo(auto l) { // C++20
    l(10);
}
```

C++ Stateless Lambda Expressions

Using lambda expressions as function parameters

```
#include <functional> // for std::function

void foo(std::function<void(int)> l) { // C++14
    l(10);
}

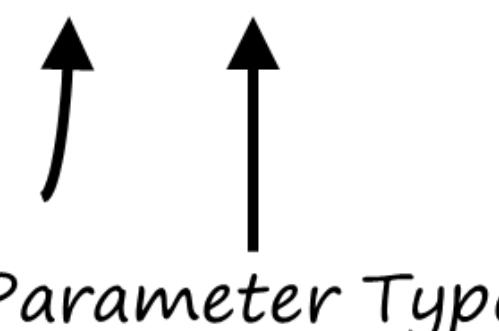
or

void foo(void (*l)(int)) { // C++14
    l(10);
}

or

void foo(auto l) { // C++20
    l(10);
}
```

Return Type *Parameter Type*



C++ Stateless Lambda Expressions

Using lambda expressions as function parameters

```
#include <functional> // for std::function

void foo(std::function<void(int)> l) { // C++14
    l(10);
}

or

void foo(void (*l)(int)) { // C++14
    l(10);
}

or

void foo(auto l) { // C++20
    l(10);
}
```

C++ Stateless Lambda Expressions

Returning lambda expressions from functions

```
#include <functional> // for std::function

std::function<void(int)> foo() {
    return [] (int x) {std::cout << x; };
}

or

void (*foo())(int) {
    return [] (int x) {std::cout << x; };
}

or

auto foo() {
    return [] (int x) {std::cout << x; };
}
```

C++ Stateless Lambda Expressions

Returning lambda expressions from functions

```
#include <functional> // for std::function

std::function<void(int)> foo() {
    return [] (int x) {std::cout << x; };
}

or

void (*foo())(int) {
    return [] (int x) {std::cout << x; };
}

or

auto foo() {
    return [] (int x) {std::cout << x; };
}
```

All 3 versions are used the same way

```
auto l = foo();
l(10); // Displays 10
```

C++ Stateless Lambda Expressions

Using lambda expressions as function parameters

```
foo( [] (int x) {std::cout << x;});  
  
or  
  
auto l = [] (int x) {std::cout << x;};  
  
foo(l);
```

C++ Stateless Lambda Expressions

Using lambda expressions as predicates

```
void print_if(std::vector<int> nums, bool (*predicate)(int)) {
    for (int i: nums)
        if (predicate(i))
            std::cout << i;
}

int main() {

    std::vector<int> nums {1,2,3};

    print_if(nums, [] (auto x) {return x % 2 == 0;}); // Displays evens
    print_if(nums, [] (auto x) {return x % 2 != 0;}); // Displays odds

    return 0;
}
```

C++ Stateful Lambda Expressions

```
[captured_variables] () -> return_type specifiers { };
```



Non-empty Capture List
Defines what information/variables should be captured

C++ Stateful Lambda Expressions

Compilation of stateless lambda expressions

Lambda definition

```
auto l = [] (int x) {std::cout << x;};
```

Compiler-generated closure

```
class CompilerGeneratedName {
public:
    CompilerGeneratedName();

    void operator() (int x) {
        std::cout << x;
    }
}
```

C++ Stateful Lambda Expressions

Compilation of stateful lambda expressions

Lambda definition

```
int y {10};

auto l = [y] (int x) {std::cout << x + y;};
```

Compiler-generated closure

```
class CompilerGeneratedName {
private:
    int y;
public:
    CompilerGeneratedName(int y) : y{y} {}

    void operator() (int x) const {
        std::cout << x + y;
    }
}
```

C++ Stateful Lambda Expressions

Capture by value

```
int x {100};

[x] () {std::cout << x;} () ;      // Displays 100
```

C++ Stateful Lambda Expressions

Using `mutable` to modify variables captured by value

```
int x {100};

[x] () mutable {
    x += 100;
    std::cout << x;           // Displays 200
}();

std::cout << x;           // Displays 100
```

C++ Stateful Lambda Expressions

Capture by reference

```
int x {100};

[ &x ] () {x += 100;} ();

std::cout << x;      // Displays 200
```

C++ Stateful Lambda Expressions

Capture by value and reference

```
[x, y]      // Capture both x and y by value
[x, &y]      // Capture x by value and y by reference
[&x, y]      // Capture x by reference and y by value
[&x, &y]      // Capture both x and y by reference
```

C++ Stateful Lambda Expressions

Default captures

```
[=]      // Default capture by value
[&]      // Default capture by reference
[this]   // Default capture this object by reference
```

C++ Stateful Lambda Expressions

Using default and explicit captures

```
[=, &x]      // Default capture by value but capture x by reference  
[&, y]       // Default capture by reference but capture y by value  
[this, z]     // Default capture this by but capture z by value
```

Bonus Section Overview

Using Visual Studio Code with C++

Visual Studio Code - VS Code

- Microsoft
- Free
- Cross-platform
- Cross-language
- Fast
- Small
- Powerful
- Extensible

Bonus Section Overview

Using Visual Studio Code with C++

- What we will do in this section
 - Install VS Code
 - Configure VS Code to work with C++
 - Setup a folder structure that allows us to use multiple projects
 - Build and run C++ projects
 - Build and debug C++ projects
 - How to use the source-code provided in the course
- We'll do this for
 - Windows
 - Mac OSX
 - Linux (Ubuntu 20.04 LTS)

Bonus Section Overview

C++ Enumerations

- What is an enumeration?
 - Motivation
- Structure of an enumeration
- Types of enumerations
 - Unscoped enumeration
 - Scoped enumeration
- Enumerations in use

C++ Enumerations

What is an enumeration?

- A user-defined type that models a set of constant integral values

C++ Enumerations

What is an enumeration?

- A user-defined type that models a set of constant integral values
 - The days of the week (Mon, Tue, Wed, . . .)
 - The months of the year (Jan, Feb, Mar, . . .)
 - The suits in a deck of cards (Clubs, Hearts, Spades, Diamonds)
 - The values in a deck of cards (Ace, Two, Three, . . .)
 - States of a system (Idle, Defense_Mode, Attack_Mode, . . .)
 - The directions on a compass (North, South, East, West)

C++ Enumerations

Motivation

- Prior to enumerated types
 - Unnamed numerical constants
 - "Magic numbers"
- These constants would be used as conditionals in control statements
- Often, one would have no idea what an algorithm was doing
- As a result, many algorithms suffered from low readability and high numbers of logic errors

Motivation

Readability

```
int state;  
std::cin >> state;  
  
if (state == 0)  
    initiate(3);  
else if (state == 1)  
    initiate(4);  
else if (state == 2)  
    initiate(5);
```

Motivation

Readability

```
enum State {Engine_Failure = 0, Inclement_Weather = 1, Nominal = 2};  
enum Sequence {Abort = 3, Hold = 4, Launch = 5};  
  
int user_input;  
std::cin >> user_input;  
State state = State(user_input);  
  
if (state == Engine_Failure) // state = 0  
    initiate(Abort); // sequence = 3  
else if (state == Inclement_Weather) // state = 1  
    initiate(Hold); // sequence = 4  
else if (state == Nominal) // state = 2  
    initiate(Launch); // sequence = 5
```

Motivation

Algorithmic correctness

```
int state = get_state();  
  
if (state == 0)  
    initiate(3);  
  
else if (state == 1)  
    initiate(4);  
  
else if (state == 2)  
    initiate(5);
```

```
int get_state() {  
    return state_of_fridge;  
}  
  
int getState() {  
    return state_of_rocket;  
}
```

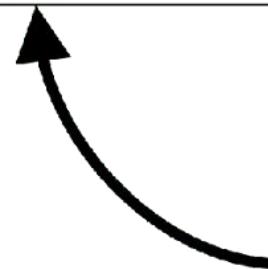
Compiles

Motivation

Algorithmic correctness

```
State state = get_state();  
  
if (state == Engine_Failure)  
    initiate(Abort);  
  
else if (state == Inclement_Weather)  
    initiate(Hold);  
  
else if (state == Nominal)  
    initiate(Launch);
```

```
int get_state() {  
    return state_of_fridge;  
}  
  
State getState() {  
    return state_of_rocket;  
}
```



Does not
compile

C++ Enumerations

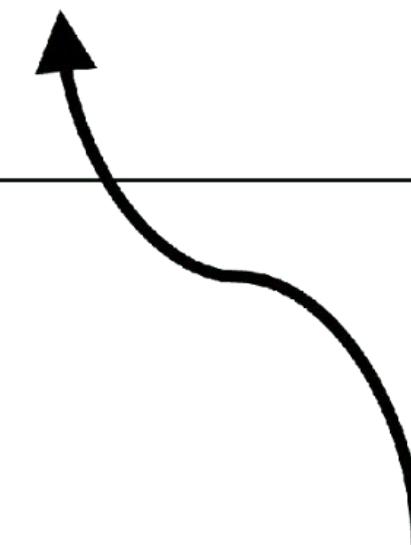
The Structure of an Enumeration

```
enum-key enum-name : enumerator-type { };
```

C++ Enumerations

The Structure of an Enumeration

```
enum-key enum-name : enumerator-type { } ;
```

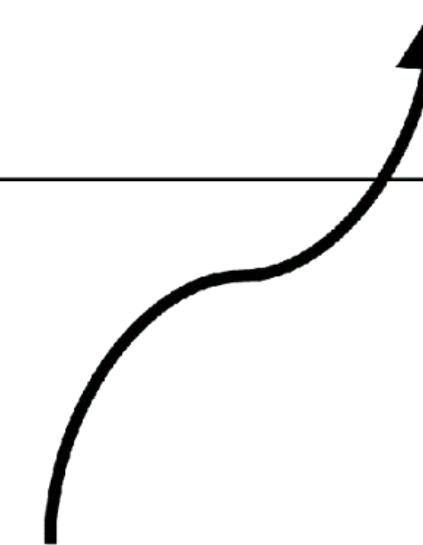


*Enum Key
Defines the scope of the enumeration*

C++ Enumerations

The Structure of an Enumeration

```
enum-key enum-name : enumerator-type { } ;
```

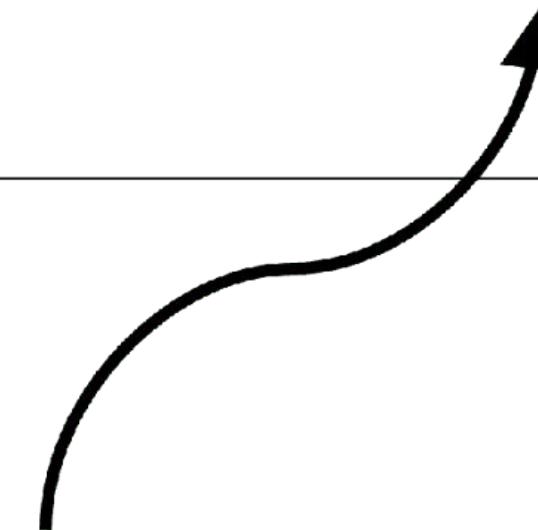


Enum Name
Optional name of the enumeration

C++ Enumerations

The Structure of an Enumeration

```
enum-key enum-name : enumerator-type { };
```

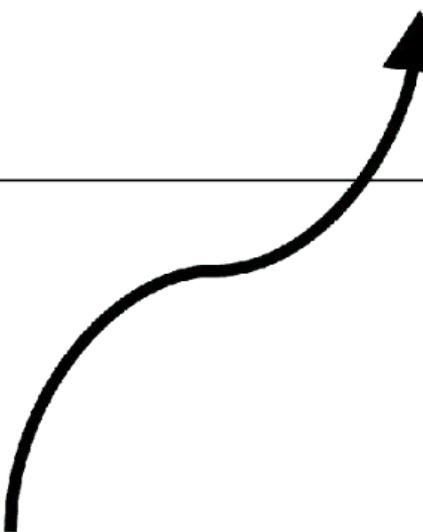


Enumerator Type
Can be omitted and the compiler
will try to deduce it!

C++ Enumerations

The Structure of an Enumeration

```
enum-key enum-name : enumerator-type { };
```



Enumerator List
List of enumerator
definitions

C++ Enumerations

Enumerator list

```
enum {Red, Green, Blue};
```

C++ Enumerations

Enumerator list

```
enum {Red, Green, Blue};           // Implicit initialization  
        0      1      2
```

C++ Enumerations

Enumerator list

```
enum {Red, Green, Blue};           // Implicit initialization  
        0      1      2  
  
enum {Red = 1, Green = 2, Blue = 3}; // Explicit initialization
```

C++ Enumerations

Enumerator list

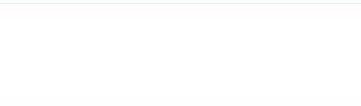
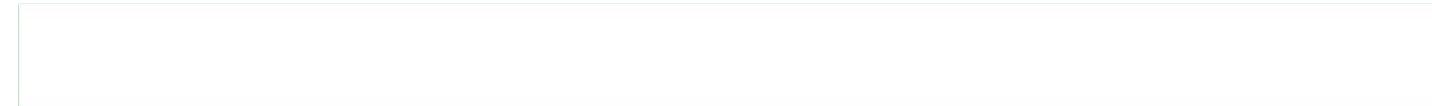
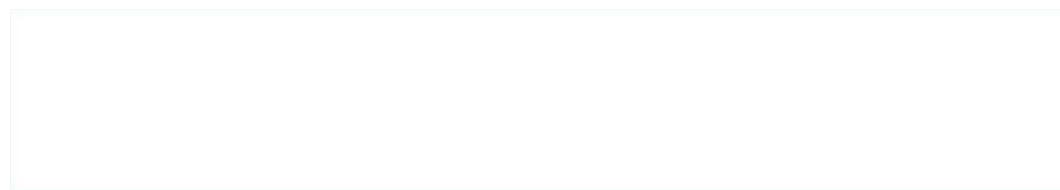
```
enum {Red, Green, Blue};           // Implicit initialization  
        0      1      2  
  
enum {Red = 1, Green = 2, Blue = 3}; // Explicit initialization  
  
enum {Red = 1, Green, Blue};        // Explicit/Implicit initialization  
        2      3
```

C++ Enumerations

Enumerator type

```
enum {Red, Green, Blue};
```

0 1 2



Integral Type	Width in bits
int	at least 16
unsigned int	
long	at least 32
unsigned long	
long long	at least 64
unsigned long long	

C++ Enumerations

Enumerator type

```
enum {Red, Green, Blue};
```

0 1 2

000 001 010

Integral Type	Width in bits
int	at least 16
unsigned int	
long	at least 32
unsigned long	
long long	at least 64
unsigned long long	

C++ Enumerations

Enumerator type

```
enum {Red, Green, Blue};           // Underlying type: int  
0    1    2  
000  001  010
```

Integral Type	Width in bits
int	at least 16
unsigned int	
long	at least 32
unsigned long	
long long	at least 64
unsigned long long	

C++ Enumerations

Enumerator type

```
enum {Red, Green, Blue};           // Underlying type: int  
0    1    2  
000  001  010  
  
enum {Red, Green, Blue = -32769};  
0    1    -32769
```

Integral Type	Width in bits
int	at least 16
unsigned int	
long	at least 32
unsigned long	
long long	at least 64
unsigned long long	

C++ Enumerations

Enumerator type

```
enum {Red, Green, Blue};           // Underlying type: int  
0    1    2  
000  001  010  
  
enum {Red, Green, Blue = -32769};  
0    1    -32769  
000  001  11111111111111011111111111111111
```

Integral Type	Width in bits
int	at least 16
unsigned int	
long	at least 32
unsigned long	
long long	at least 64
unsigned long long	

C++ Enumerations

Enumerator type

```
enum {Red, Green, Blue};           // Underlying type: int
    0   1   2
    000 001 010

enum {Red, Green, Blue = -32769}; // Underlying type: long
    0   1   -32769
    000 001 11111111111111011111111111111111
```

Integral Type	Width in bits
int	at least 16
unsigned int	
long	at least 32
unsigned long	
long long	at least 64
unsigned long long	

C++ Enumerations

Enumerator type

```
enum {Red, Green, Blue};           // Underlying type: int
      0   1   2
      000 001 010

enum {Red, Green, Blue = -32769}; // Underlying type: long
      0   1   -32769
      000 001 11111111111111011111111111111111

enum : uint8_t {Red, Green, Blue}; // Underlying type: unsigned 8 bit int

enum : long long {Red, Green, Blue}; // Underlying type: long long
```

Integral Type	Width in bits
int	at least 16
unsigned int	
long	at least 32
unsigned long	
long long	at least 64
unsigned long long	

C++ Enumerations

Enumeration name

```
enum {Red, Green, Blue};  
  
int my_color;  
  
my_color = Green;      // Valid  
  
my_color = 4;          // Also valid
```

C++ Enumerations

Enumeration name

*Anonymous
No type safety*

```
enum { Red, Green, Blue };

int my_color;

my_color = Green;      // Valid

my_color = 4;          // Also valid
```

C++ Enumerations

Enumeration name

Anonymous
No type safety

```
enum {Red, Green, Blue};
```

```
int my_color;
```

```
my_color = Green; // Valid
```

```
my_color = 4; // Also valid
```

Named
Type safe

```
enum Color {Red, Green, Blue};
```

```
Color my_color;
```

```
my_color = Green; // Valid
```

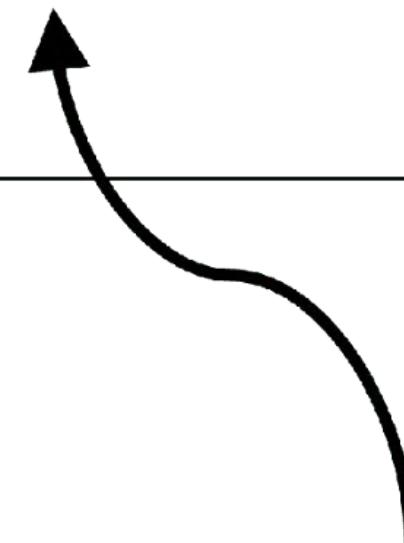
```
my_color = 4; // Not valid
```

C++ Unscoped Enumerations

```
enum-key enum-name : enumerator-type { } ;
```

C++ Unscoped Enumerations

```
enum-key enum-name : enumerator-type { };
```



Enum Key
Defines the scope of the enumeration

C++ Unscoped Enumerations

```
enum enum-name : enumerator-type { };
```



Enum

Defines an unscoped enumeration

C++ Unscoped Enumerations

Using if and switch statements with unscoped enumerations

```
State state = get_state();  
  
if (state == Nominal)  
    initiate(Launch);  
  
else if (state == Inclement_Weather)  
    initiate(Hold);  
  
else if (state == Engine_Failure)  
    initiate(Abort);
```

C++ Unscoped Enumerations

Using if and switch statements with unscoped enumerations

```
State state = get_state();

if (state == Nominal)                                // Accessed first
    initiate(Launch);

else if (state == Inclement_Weather)
    initiate(Hold);

else if (state == Engine_Failure)
    initiate(Abort);
```

C++ Unscoped Enumerations

Using if and switch statements with unscoped enumerations

```
State state = get_state();

if (state == Nominal)                                // Accessed first
    initiate(Launch);

else if (state == Inclement_Weather)                 // Accessed second
    initiate(Hold);

else if (state == Engine_Failure)
    initiate(Abort);
```

C++ Unscoped Enumerations

Using if and switch statements with unscoped enumerations

```
State state = get_state();

if (state == Nominal)                                // Accessed first
    initiate(Launch);

else if (state == Inclement_Weather)                 // Accessed second
    initiate(Hold);

else if (state == Engine_Failure)                    // Accessed last
    initiate(Abort);
```

C++ Unscoped Enumerations

Using if and switch statements with unscoped enumerations

```
State state = get_state();

switch (state) {
    case Engine_Failure:          // Equal access time
        initiate(Abort);
        break;
    case Inclement_Weather:       // Equal access time
        initiate(Hold);
        break;
    case Nominal:                 // Equal access time
        initiate(Launch);
        break;
}
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
enum State {Engine_Failure, Inclement_Weather, Nominal};

State state;

std::cin >> state;
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
enum State {Engine_Failure, Inclement_Weather, Nominal};

State state;

std::cin >> state;      // Not allowed using standard extraction operator
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
enum State {Engine_Failure, Inclement_Weather, Nominal};      // Underlying type: int
                                                               0           1           2
std::underlying_type_t<State> user_input;
std::cin >> user_input;
State state = State(user_input);
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
enum State {Engine_Failure, Inclement_Weather, Nominal};      // Underlying type: int
                                                               0           1           2
std::underlying_type_t<State> user_input;                      // Type: int
std::cin >> user_input;
State state = State(user_input);
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
enum State {Engine_Failure, Inclement_Weather, Nominal};      // Underlying type: int
                                                               0           1           2
std::underlying_type_t<State> user_input;                      // Type: int
std::cin >> user_input;                                         // userInput = 3
State state = State(user_input);                                // state = 3
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
enum State {Engine_Failure, Inclement_Weather, Nominal};      // Underlying type: int
                                                               0           1           2
std::underlying_type_t<State> user_input;                      // Type: int
std::cin >> user_input;
State state;
switch (user_input) {
    case Engine_Failure:          state = State(user_input); break;
    case Inclement_Weather:       state = State(user_input); break;
    case Nominal:                 state = State(user_input); break;
    default:                      std::cout << "User input is not a valid state.";
}
}
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
enum State {Engine_Failure, Inclement_Weather, Nominal};      // Underlying type: int

std::istream& operator>>(std::istream& is, State& state) {
    std::underlying_type_t<State> user_input;                      // Type: int
    is >> user_input;
    switch (user_input) {
        case Engine_Failure:           state = State(user_input); break;
        case Inclement_Weather:        state = State(user_input); break;
        case Nominal:                 state = State(user_input); break;
        default:                      std::cout << "User input is not a valid state.";
    }
    return is;
}
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
State state;  
  
std::cin >> state;      // Valid with overloaded extraction operator
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
enum State {Engine_Failure, Inclement_Weather, Nominal};

State state = Engine_Failure;

std::cout << state;      // Displays 0
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
enum State {Engine_Failure, Inclement_Weather, Nominal};

State state = Engine_Failure;

switch (state) {                                // Displays "Engine Failure"
    case Engine_Failure:                      std::cout << "Engine Failure"; break;
    case Inclement_Weather:                   std::cout << "Inclement Weather"; break;
    case Nominal:                            std::cout << "Nominal"; break;
    default:                                std::cout << "Unknown";
}
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
enum State {Engine_Failure, Inclement_Weather, Nominal};

std::ostream& operator<<(std::ostream& os, State& state) {
    switch (state) {
        case Engine_Failure:          os << "Engine Failure"; break;
        case Inclement_Weather:       os << "Inclement Weather"; break;
        case Nominal:                 os << "Nominal"; break;
        default:                      os << "Unknown";
    }
    return os;
}
```

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
enum State {Engine_Failure, Inclement_Weather, Nominal}

std::ostream& operator<<(std::ostream& os, State& state) {
    switch (state) {
        case Engine_Failure:          os << "Engine Failure"; break;
        case Inclement_Weather:       os << "Inclement Weather"; break;
        case Nominal:                 os << "Nominal"; break;
        default:                      os << "Unknown";
    }
    return os;
}
```

This should be a
const reference

C++ Unscoped Enumerations

Using cin and cout with unscoped enumerations

```
State state = Engine_Failure;  
std::cout << state; // Displays "Engine Failure"
```

C++ Scoped Enumerations

```
enum enum-name : enumerator-type { };
```



*Enum
Defines an unscoped enumeration*

C++ Scoped Enumerations

```
enum class enum-name : enumerator-type { };
```



enum class or enum struct
Defines a scoped enumeration

C++ Scoped Enumerations

Motivation

```
enum Whale {Blue, Beluga, Gray};  
enum Shark {Greatwhite, Hammerhead, Bull};
```

C++ Scoped Enumerations

Motivation

```
enum Whale {Blue, Beluga, Gray};  
  
enum Shark {Greatwhite, Hammerhead, Bull};  
  
if (Beluga == Hammerhead)  
    std::cout << "A beluga whale is equivalent to a hammerhead shark.";
```

C++ Scoped Enumerations

Motivation

```
enum Whale {Blue, Beluga, Gray};  
  
enum Shark {Greatwhite, Hammerhead, Bull, Blue};  
  
if (Beluga == Hammerhead)  
    std::cout << "A beluga whale is equivalent to a hammerhead shark.";
```

C++ Scoped Enumerations

Motivation

```
enum Whale {Blue, Beluga, Gray};

enum Shark {Greatwhite, Hammerhead, Bull, Blue};      // Error! (Blue already defined)

if (Beluga == Hammerhead)
    std::cout << "A beluga whale is equivalent to a hammerhead shark.";
```

C++ Scoped Enumerations

Using if and switch statements with scoped enumerations

```
enum class Whale {Blue, Beluga, Gray};

Whale whale = Whale::Beluga;

if (whale == Whale::Blue)
    std::cout << "Blue whale";
else if (whale == Whale::Beluga)
    std::cout << "Beluga whale";
else if (whale == Whale::Gray)
    std::cout << "Gray whale";
```

```
enum class Whale {Blue, Beluga, Gray};

Whale whale = Whale::Beluga;

switch (whale) {
    case Whale::Blue:
        std::cout << "Blue whale"; break;
    case Whale::Beluga:
        std::cout << "Beluga whale"; break;
    case Whale::Gray:
        std::cout << "Blue whale"; break;
}
```

C++ Scoped Enumerations

Using scoped enumerator values

```
enum class Item {Milk = 350, Bread = 250, Apple = 132};      // Underlying type: int

int milk_code = Item::Milk;
int total = Item::Milk + Item::Bread;
std::cout << Item::Milk;
```

C++ Scoped Enumerations

Using scoped enumerator values

```
enum class Item {Milk = 350, Bread = 250, Apple = 132};      // Underlying type: int

int milk_code = Item::Milk;                                // Error! (Cannot convert Item to int)
int total = Item::Milk + Item::Bread;                        // Error! (No matching '+' operator)
std::cout << Item::Milk;                                    // Error! (No matching '<<' operator)
```

C++ Scoped Enumerations

Using scoped enumerator values

```
enum class Item {Milk = 350, Bread = 250, Apple = 132};      // Underlying type: int

int milk_code = int(Item::Milk);                                // milk_code = 350
or
int milk_code = static_cast<int>(Item::Milk);

int total = int(Item::Milk) + int(Item::Bread);                  // total = 600
std::cout << underlying_type_t<Item>(Item::Milk);            // Displays 350
```