Object-oriented Programming in C++

Stephane Redon

OneAngstrom
https://www.oneangstrom.com
About this document

- This document is an introduction to C++ provided by OneAngstrom to help you kickstart development of modules for SAMSON, the open platform for molecular modeling available at https://www.samson-connect.net.

- This introduction can be read as a standalone, but the best way to learn is probably to head to the SAMSON developer guide at https://s-c.io/dg and use the present introduction to C++ when necessary.
Overview

• C++ is a compiled language: a program written in C++...

```cpp
int main(int argc, char *argv[]) {
    // compute 10!
    int n=1;
    for (int i=1;i<10;i++) n=n*i;
    return n;
}
```

⚠️ We will explain this program soon...
Overview

- C++ is a **compiled language**: ... is turned into machine language

```c
int main(int argc, char *argv[]) {
    push ebp
    mov ebp,esp
    sub esp,0D8h
    push ebx
    push esi
    push edi
    lea edi,[ebp-0D8h]
    mov ecx,36h
    mov eax,0CCCCCCCCh
    rep stos dword ptr es:[edi]

    // compute 10!
    int n=1;
    mov dword ptr [n],1
    for (int i=1;i<10;i++) n=n*i;
    mov dword ptr [ebp-14h],1
    jmp main+37h (01311307h)
    mov eax,dword ptr [ebp-14h]
    add eax,1
    mov dword ptr [ebp-14h],eax
    cmp dword ptr [ebp-14h],0Ah
    jge main+49h (013113E9h)
    mov eax,dword ptr [n]
    imul eax,dword ptr [ebp-14h]
    mov dword ptr [n],eax
    jmp main+2Eh (013113CEh)

    return n;
    mov eax,dword ptr [n]
}
```

⚠️ ... but not this one ;-)
Memory
Memory

- Memory can be modeled as a linear array

One bit (0 or 1)
Memory

- Memory can be modeled as a linear array

8 bits = 1 byte
1024 bytes = 1 kilobyte = 1 KB
1024 KB = 1 megabyte = 1 MB
1024 MB = 1 gigabyte = 1 GB
1024 GB = 1 terabyte = 1 TB
...
Memory

- The memory cells have **addresses**

- On a 32-bit system, memory addresses are coded on 32 bits (4 bytes)
  - Maximum 4GB

- On a 64-bit system, memory addresses are coded on 64 bits (8 bytes)
  - Maximum 16 exbibytes = \(2^{64}\) bytes = 1 billion of gigabytes
Memory

- An executable uses three memory segments
  - Code segment
    - Stores the compiled program (the assembly code)
  - Stack segment
    - Memory allocated for functions
    - Function-level memory only lasts as long as the function lasts
  - Heap segment
    - Memory allocated for persistent data
    - Lasts until de-allocation, or when the executable ends

- More on this later
First programs
The minimal program

The main function (has to be called main)

Return type (has to be int)

```
int main() {
    return 0;
}
```
The traditional Hello world! program

#include <iostream>

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

// This is a one-line comment

/* This is the first line of a multi-line comment
   This is the second line of a multi-line comment */

std::cout << "Hello" << std::endl; // comments may be placed after a statement

std::cout << "Hello" << /* and even inside a statement (please don't) */ std::endl;

/* comments may be used to de-activate parts of code
std::cout << "Hello" << std::endl;
*/
Comments

- Extremely important

- Used to document the code
  - Variables
  - Files
  - Functions
  - Algorithms
  - ...

- Think about your collaborators

- Think about your future you
Fundamental data types
Fundamental data types

- **bool** (one bit): true or false
  
  ```
  bool b=true;
  ```

- **char** (one byte): integer between -128 and 127
  
  ```
  char c='A';
  char d=65; // 65 is the ASCII code of 'A', so d=c
  ```

- **short** (usually two bytes): integer between -32,768 and 32,767
  
  ```
  short s=1203;
  ```

- **int** (usually four bytes): integer between -2,147,483,648 and 2,147,483,647
  
  ```
  int i=0;
  ```

- **long** (usually eight bytes): –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
  
  ```
  long l=12012930;
  ```

- **char**, **int** and **long** can be prefixed by ‘unsigned’
  
  ```
  unsigned int i=4294967295; // maximum unsigned int
  unsigned short s=-1; // s=65535
  ```
Fundamental data types

- **float** (four bytes): floating-point value (single precision)

```cpp
float f1=1.0; // f1 is set to 1.0
float f2=1.0000000000000001; // f2 is also set to 1.0!
```

- **double** (eight bytes): floating-point value (double precision)

```cpp
double d1=1.0; // d1 is set to 1.0
double d2=1.0000000000000001; // d2 is set to 1.0000000000000001
double d3=1.0000000000000001; // d3 is set to 1.0!
```

- Declaration and assignment may be separate

```cpp
int i;
int i=2;
```

- C++ has rules for conversions between types

- **void**: the type that represents nothing (more on this later)
Arrays
Arrays

- Structure

```cpp
typeName variableName[size];
```

- Arrays are **not** initialized (saves time)

- Example

```cpp
int i[10]; // an array of 10 integers
i[4]=65;   // affects 65 to the 5th element
std::cout << i[4] << std::endl; // prints 65
```

- In memory, array elements are contiguous
Streams
Streams

- We will use them to print and read user input

```cpp
int i; // i is an integer
double k; // k is a double
unsigned char a[4]; // a is (a pointer to the first element of) an array of unsigned char
double d; // d is a double

std::cin >> i; // affects to i the first number entered by the user
std::cin >> k; // affects to k the second number entered by the user
std::cin >> a; // affects to a the chars entered by the user
std::cin >> d; // affects to k the third number entered by the user

std::cout << i << std::endl; // prints the first number entered by the user
std::cout << k << std::endl; // prints the second number entered by the user
std::cout << a << std::endl; // prints the chars entered by the user
std::cout << d << std::endl; // prints the third number entered by the user
```
A slightly more friendly Hello world! program

Include the string header file

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

```cpp
int main() {
    std::cout << "Hello! Please enter your name: ";

    std::string name; // name is a string variable
    std::cin >> name;

    std::cout << "Hello " << name << "!" << std::endl;

    return 0;
}
```

Standard input stream
(more on this later)
References and pointers
References and pointers

- Variables are stored in memory, at a given address

```
unsigned char i=165;
```

The variable `i` is stored at address 17

```
... 0 1 0 1 1 0 1 0 0 1 0 1 1 1 1 0 1 ...
```

Address ‘17 bytes’

Address ‘18 bytes’
References and pointers

- A reference is a **supplementary name for a variable**

  The reference **has to be assigned immediately**

  Note the &

- Syntax: `typeName& referenceName=variableName;`

  referenceName is a **supplementary name for the variable** with name ‘variableName’
Variables are stored in memory, at a given address.

```c
unsigned char i = 165;
unsigned char& r = i;
```

The variable `i` is stored at address 17.

`r` is just another name for `i`.

References and pointers.
References and pointers

- A pointer is a **variable which contains the address of another variable**

Note the *

- Declaration syntax: `typeName* pointerName;`

  `pointerName` is a **variable which contains the address of another variable** of type `typeName`

- The address of a variable is obtained with `&`
Variables are stored in memory, at a given **address**

```cpp
unsigned char i = 165;
unsigned char* p = &i;
```

The variable `i` is stored at address 17
The variable `p` contains 17

![Address '17 bytes'](image)

![Address '18 bytes'](image)
References and pointers

- Variables are stored in memory, at a given address

```c
unsigned char i=165;                   // The variable i is stored at address 17
unsigned char* p=&i;                   // The variable p contains 17
```

---

```
unsigned char* p; // *p is an unsigned char
unsigned char* p; // *p is an unsigned char
```
References and pointers

- Variables are stored in memory, at a given **address**

```cpp
unsigned char i=165;
unsigned char* p=&i;
```

The variable `i` is stored at address 17
The variable `p` contains 17

---

Dereferences (“depoints”) pointer `p`

```cpp
*p=65;
std::cout << *p << std::endl;  // affects 65 to i
```

// prints A
References and pointers

• Variables are stored in memory, at a given **address**

```c
unsigned char i = 165;
unsigned char* p = &i;
```

The variable `i` is stored at address 17
The variable `p` contains 17

---

```
0 1 0 1 1 0 1 0 0 1 0 1 1 1 0 1
```

Address ‘17 bytes’

`p` points to the variable `i`

Address ‘18 bytes’
• Variables are stored in memory, at a given **address**

```
unsigned char i=165;
unsigned char* p=&i;
```

The variable `i` is stored at address 17
The variable `p` contains 17

The variable `p` is stored elsewhere, for example at address 23

References and pointers

Address ‘17 bytes’

Address ‘18 bytes’

Address ‘23 bytes’

Address ‘24 bytes’
References and pointers

- Variables are stored in memory, at a given **address**

```c
unsigned char i=165;
unsigned char* p=&i;
```

The variable `i` is stored at address 17
The variable `p` contains 17

The variable `p` is stored elsewhere, for example at address 23

- We can perform **pointer arithmetics**

Address ‘17 bytes’
Address ‘18 bytes’
• Variables are stored in memory, at a given **address**

```c
unsigned char i = 165;
unsigned char* p = &i;
```

The variable `i` is stored at address 17
The variable `p` contains 17
References and pointers

- Variables are stored in memory, at a given **address**

  ```
  unsigned char i = 165;
  unsigned char* p = &i;
  ```

  The variable `i` is stored at address 17
  The variable `p` contains 17

```
... 0 1 0 1 1 0 1 0 0 1 0 1 1 1 0 1 ...
```

Address ‘17 bytes’

Address ‘18 bytes’

```
p++;
```
References and pointers

- Variables are stored in memory, at a given **address**

  ```
  unsigned char i=165;
  unsigned char* p=&i;
  ```

  The variable i is stored at address 17
  The variable p contains 17

  ```
  p--;  
  ```

  Address ‘17 bytes’
  Address ‘18 bytes’
References and pointers

• A variable representing an array “is” a pointer to the first element of the array

```
unsigned char i[10];  // i is a pointer to the first element
unsigned char* p = i; // p is equal to i
```
References and pointers

- A variable representing an array "is" a pointer to the first element of the array

```c
unsigned char i[10]; // i is a pointer to the first element
unsigned char* p=i; // p is equal to i
```

```c
i[4]=65; // affects 65 to the 5th element
std::cout << i[4] << std::endl; // prints A
```
A variable representing an array “is” a pointer to the first element of the array.

```c
unsigned char i[10]; // i is a pointer to the first element
unsigned char* p=i;  // p is equal to i
```

```
p[4]=65;  // affects 65 to the 5th element
std::cout << i[4] << std::endl;  // prints A
```
References and pointers

- A variable representing an array "is" a pointer to the first element of the array.

```cpp
unsigned char i[10]; // i is a pointer to the first element
unsigned char* p=i; // p is equal to i
```

Address '17 bytes'

Address '18 bytes'

```cpp
*(p+4)=65; // affects 65 to the 5th element
std::cout << i[4] << std::endl; // prints A
```
Tests
Tests

• Structure

```cpp
if (condition_statement) {
    statements
}
else {
    statements
}
```

The else block is optional
Tests

// check whether i is equal to 2

if (i==2) {
    std::cout << "i is equal to 2" << std::endl;
}
else {
    std::cout << "i is not equal to 2" << std::endl;
}
Tests

```cpp
if ((i==2) && (j==3)) { // && is the 'and' operator
    std::cout << "i is equal to 2" << std::endl;
    std::cout << "j is equal to 3" << std::endl;
}
else {
    std::cout << "i is not equal to 2" << std::endl;
    std::cout << "or j is not equal to 3" << std::endl;
}
```
if ((i==2)&&(j==3)) { // evaluate both clauses before concluding
    std::cout << "i is equal to 2" << std::endl;
    std::cout << "j is equal to 3" << std::endl;
}
else {
    std::cout << "i is not equal to 2" << std::endl;
    std::cout << "or j is not equal to 3" << std::endl;
}
Tests

```cpp
int i=3;
int j=3;

// test values

bool a=((i==2)&&(j==3)); // and: a is false
bool b=((i==2)||(j==3)); // or : b is true
bool c=!((i==2)); // not: c is true;
```
Loops
The for loop

- Structure

```cpp
for (initial_statement; continuation_statement; iteration_statement) {
    statements
}
```
The for loop

// print hello ten times
for (int i=0;i<10;i++) std::cout << "Hello" << std::endl;

// print hello ten times
for (int i=10;i>0;i--) std::cout << "Hello" << std::endl;

// print hello ten times
for (int i=0;i<100;i=i+10) std::cout << "Hello" << std::endl;

// print hello ten times
for (double d=345.3;d<322370.45;d=-2.0*d+17.0875) std::cout << "Hello" << std::endl;
The for loop

```cpp
// print hello ten times (no initial statement)

int i=0;

for (;i<10;i++) std::cout << "Hello" << std::endl;

// print hello ten times (no condition statement)

for (int i=0;i++)
{
    if (i>=10) break; // if i>=10, exit the for loop
    std::cout << "Hello" << std::endl;
}

// print hello ten times (no iteration statement)

for (int i=0;i<10;)
{
    std::cout << "Hello" << std::endl;
    i++;
}
```
// print hello ten times (no statements)

int i=0;

for (;;) {
    if (i>=10) break; // if i>=10, exit the for loop
    std::cout << "Hello" << std::endl;
    i++;
}

The for loop

// print hello ten times (imbricated for loops)

for (int i=0; i<2; i++) {
    for (int j=0; j<5; j++) std::cout << "Hello" << std::endl;
}

The while loop

• Structure

```
while (continuation_statement) {
    statements
}
```
The while loop

// print hello ten times

int i=0;

while (i<10) {
    std::cout << "Hello" << std::endl;
    i++;
}

The do-while loop

- Structure

```cpp
do {
    statements
} while (condition_statement)
```

Note: at least one iteration is performed!
The do-while loop

```cpp
// print hello ten times

int i=0;

do {
    std::cout << "Hello" << std::endl;
    i++;
} while (i<10);
```
Functions
Functions

- Functions are used to **organize large programs**, and **avoid repetition**
- Structure

```cpp
returnTypeName functionName(arguments) {

    statements

}
```

- A function receives **zero or more arguments**
- A function has a **single return type** (may be ‘`void’`)  
- When a function has to return something, the `return` keyword is used
- A function may **call other functions**, and even **call itself** (“recursive function”)  
- Two functions with the same name but different arguments can **co-exist**
Functions

This function returns nothing

```cpp
void printHello() {
    std::cout << "Hello" << std::endl;
}
```

Even though the function **returns nothing**, it still **prints** something on the console.
Functions

This function receives one argument, a double called \( d \)

```cpp
void printNumber(double d) {
    std::cout << d << std::endl;
}
```

Even though the function returns nothing, it still prints something on the console.
This function returns a double

This function receives one argument, a double called d

```cpp
double multiplyByTwo(double d) {
    double result = 2 * d;
    return result;
}
```

This time, the function prints nothing, but it returns a double
In this version, the unnecessary temporary variable has been removed

```cpp
double multiplyByTwo(double d) {
    return 2*d;
}
```

This function receives one argument, a double called d.
This function calls two functions to perform its task.
void printNumber(double d) {
    std::cout << d << std::endl;
}

double multiplyByTwo(double d) {
    return 2*d;
}

void printNumberMultipliedByTwo(double d) {
    printNumber(multiplyByTwo(d));
}

This function calls two functions to perform its task.

In this version, the unnecessary temporary variable has been removed.
This function sometimes calls itself, it is thus **recursive**

```cpp
int factorial(int n) {
    if (n<=1) return 1;
    return n*factorial(n-1);
}
```

Stopping condition

General case
Functions

First overloaded version

```cpp
int multiplyByTwo(int i) {
    return 2*i;
}
```

Arguments must differ

Second overloaded version

```cpp
double multiplyByTwo(double d) {
    return 2*d;
}
```

(more on this in Session 3)
Argument passing
Argument passing

- Functions may receive arguments by value or by reference

- When an argument is passed by value, a copy of it is made

```c++
void increment(int a) {
    a++;
}

int main() {
    int i=65; // affects 65 to i
    increment(i); // pass i by value to increment
    std::cout << i << std::endl; // prints 65

    return 0;
}
```
Argument passing

- Functions may receive arguments by value or by reference

- When an argument is passed by value, a copy of it is made

```cpp
void increment(int i) {
    i++; // i is a copy of the passed argument, even if it has the same name
}

int main() {
    int i=65; // affects 65 to i
    increment(i); // pass i by value to increment
    std::cout << i << std::endl; // prints 65
    return 0;
}
```
**Argument passing**

- Functions may receive arguments **by value** or **by reference**

- When an argument is **passed by reference**, it may be accessed in the function

```cpp
void increment(int& r) {
    r++;  // r is a reference to the passed argument (note the &)
}

int main() {
    int i=65;
    increment(i);  // affects 65 to i
    std::cout << i << std::endl;  // pass i by value to increment
    return 0;
}
```
Argument passing

- Functions may receive arguments **by value** or **by reference**

- When an argument is **passed by reference**, it may be accessed in the function

```cpp
void increment(int& i) {
    i++;
}

int main() {
    int i=65;
    increment(i);  // affects 65 to i
    std::cout << i << std::endl;  // pass i by value to increment
    return 0;
}
```

- `i` is a **reference** to the passed argument (note the `&`)
The stack
The stack

- Second type of memory segment associated to an executable
- The stack is used for temporary memory storage:
  - variables declared in the function
  - arguments passed by value to functions
The stack

- Second type of memory segment associated to an executable
- The stack is used for temporary memory storage:
  - variables declared in the function
  - arguments passed by value to functions

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}
```

How many variables are stored in the stack?

```cpp
int main() {
    int i = 65; // affects 65 to i
    i = addToInteger(i, 2); // add 2 to i
    std::cout << i << std::endl; // prints 67
    return 0;
}
```
The stack

- Second type of memory segment associated to an executable
- The stack is used for temporary memory storage:
  - variables declared in the function
  - arguments passed by value to functions

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}
```

```cpp
int main() {
    int i = 65;  // affects 65 to i
    i = addToInteger(i, 2);  // add 2 to i
    std::cout << i << std::endl;  // prints 67
    return 0;
}
```
The stack

- Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65;  // affects 65 to i
    i = addToInteger(i, 2);  // add 2 to i
    std::cout << i << std::endl;  // prints 67

    return 0;
}
```
The stack

- Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65;  // affects 65 to i
    i = addToInteger(i, 2);  // add 2 to i
    std::cout << i << std::endl;  // prints 67
    return 0;
}
```
The stack

- Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65;  // affects 65 to i
    i = addToInteger(i, 2);  // add 2 to i
    std::cout << i << std::endl;  // prints 67
    return 0;
}
```

The stack

- Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65; // affects 65 to i
    i = addToInteger(i, 2); // add 2 to i
    std::cout << i << std::endl; // prints 67
    return 0;
}
```

The stack
The stack

- Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65;          // affects 65 to i
    i = addToInteger(i, 2); // add 2 to i
    std::cout << i << std::endl; // prints 67
    return 0;
}
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>addToInteger</td>
<td>increment</td>
</tr>
<tr>
<td>main</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>65</td>
</tr>
</tbody>
</table>
The stack

- Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65; // affects 65 to i
    i = addToInteger(i, 2); // add 2 to i
    std::cout << i << std::endl; // prints 67
    return 0;
}
```

<table>
<thead>
<tr>
<th>addToInteger</th>
<th>increment</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>i</td>
<td>65</td>
</tr>
</tbody>
</table>

The stack
The stack

- Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65;  // affects 65 to i
    i = addToInteger(i, 2);  // add 2 to i
    std::cout << i << std::endl;  // prints 67
    return 0;
}
```

The stack

<table>
<thead>
<tr>
<th>addToInteger</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>addToInteger</th>
<th>increment</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>i</td>
<td>65</td>
</tr>
</tbody>
</table>
The stack

- Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65;  // affects 65 to i
    i = addToInteger(i, 2);  // add 2 to i
    std::cout << i << std::endl;  // prints 67
    return 0;
}
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>addToInteger</td>
<td></td>
<td>67</td>
</tr>
<tr>
<td>increment</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>65</td>
<td></td>
</tr>
</tbody>
</table>

The stack
The stack

• Example execution

```cpp
int addToInteger(int &i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65; // affects 65 to i
    i = addToInteger(i, 2); // add 2 to i
    std::cout << i << std::endl; // prints 67
    return 0;
}
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>addToInteger</td>
<td>result</td>
<td>67</td>
</tr>
<tr>
<td>addToInteger</td>
<td>increment</td>
<td>2</td>
</tr>
<tr>
<td>main</td>
<td>i</td>
<td>65</td>
</tr>
</tbody>
</table>
The stack

- Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65;  // affects 65 to i
    i = addToInteger(i, 2);  // add 2 to i
    std::cout << i << std::endl;  // prints 67
    return 0;
}
```
• Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65;  // affects 65 to i
    i = addToInteger(i, 2);  // add 2 to i
    std::cout << i << std::endl;  // prints 67
    return 0;
}
```
The stack

- Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65; // affects 65 to i
    i = addToInteger(i, 2);  // add 2 to i
    std::cout << i << std::endl; // prints 67

    return 0;
}
```

The stack
The stack

- Example execution

```cpp
int addToInteger(int& i, int increment) {
    int result = i + increment;
    return result;
}

int main() {
    int i = 65;  // affects 65 to i
    i = addToInteger(i, 2);  // add 2 to i
    std::cout << i << std::endl;  // prints 67
    return 0;
}
```
The heap
The heap

- Third type of memory segment associated to an executable
- The heap is used to allocate memory that persists between functions
- In C++, the `operator new` is used to allocate on the heap
- In C++, the `operator delete` is used to de-allocate from the heap

```cpp
int main() {
    int* i = new int; // i points to an int in the heap
    *i = 65; // affects 65 to *i

    char* a = new char[10]; // a points to an array of chars in the heap
    a[4] = 'A'; // affects 'A' to the 5th element of the array

    delete i; // free the memory pointed by i
    delete[] a; // free the memory pointed by a

    return 0;
}
```

Who is on the stack? Who is in the heap?
The heap

• Example execution

```cpp
int main() {

    // i points to an int in the heap
    int* i = new int;

    // affects 65 to *i
    *i = 65;

    // a points to an array of chars in the heap
    char* a = new char[10];

    // affects 'A' to the 5th element of the array
    a[4] = 'A';

    // free the memory pointed by i
    delete i;

    // free the memory pointed by a
    delete[] a;

    return 0;
}
```
The heap

• Example execution

```cpp
int main() {
    // i points to an int in the heap
    int* i = new int;

    // affects i to *i
    *i = 65;

    // a points to an array of chars in the heap
    char* a = new char[10];

    // affects 'A' to the 5th element of the array
    a[4] = 'A';

    // free the memory pointed by i
    delete i;

    // free the memory pointed by a
    delete[] a;

    return 0;
}
```
The heap

- Example execution

```cpp
int main() {
    // i points to an int in the heap
    int* i = new int;
    // affects 65 to *i
    *i = 65;
    // a points to an array of chars in the heap
    char* a = new char[10];
    // affects 'A' to the 5th element of the array
    a[4] = 'A';
    // free the memory pointed by i
    delete i;
    // free the memory pointed by a
    delete[] a;
    return 0;
}
```
The heap

- Example execution

```cpp
int main() {

    // i points to an int in the heap
    int* i = new int;

    // affects 65 to *i
    *i = 65;

    // a points to an array of chars in the heap
    char* a = new char[10];

    // affects 'A' to the 5th element of the array
    a[4] = 'A';

    // free the memory pointed by i
    delete i;

    // free the memory pointed by a
    delete[] a;

    return 0;
}
```
The heap

- Example execution

```c++
int main() {
    // i points to an int in the heap
    int* i = new int;
    // affects 65 to *i
    *i = 65;

    // a points to an array of chars in the heap
    char* a = new char[10];
    // affects 'A' to the 5th element of the array
    a[4] = 'A';

    // free the memory pointed by i
    delete i;
    // free the memory pointed by a
    delete[] a;

    return 0;
}
```
The heap

• Example execution

```cpp
int main() {
    // i points to an int in the heap
    int* i = new int;

    // affects 65 to *i
    *i = 65;

    // a points to an array of chars in the heap
    char* a = new char[10];

    // affects 'A' to the 5th element of the array
    a[4] = 'A';

    // free the memory pointed by i
    delete i;

    // free the memory pointed by a
    delete[] a;

    return 0;
}
```
The heap

- Example execution

```c
int main() {
    // i points to an int in the heap
    int* i = new int;

    // affects 65 to *i
    *i = 65;

    // a points to an array of chars in the heap
    char* a = new char[10];

    // affects 'A' to the 5th element of the array
    a[4] = 'A';

    // free the memory pointed by i
    delete i;

    // free the memory pointed by a
    delete[] a;

    return 0;
}
```
The heap

- Example execution

```cpp
int main() {
    // i points to an int in the heap
    int* i = new int;

    // affects 65 to *i
    *i = 65;

    // a points to an array of chars in the heap
    char* a = new char[10];

    // affects 'A' to the 5th element of the array
    a[4] = 'A';

    // free the memory pointed by i
    delete i;

    // free the memory pointed by a
    delete[] a;

    return 0;
}
```
The heap

• Example execution

```c++
int main() {
    // i points to an int in the heap
    int* i = new int;

    // affects 65 to *i
    *i = 65;

    // a points to an array of chars in the heap
    char* a = new char[10];

    // affects 'A' to the 5th element of the array
    a[4] = 'A';

    // free the memory pointed by i
    delete i;

    // free the memory pointed by a
    delete[] a;

    return 0;
}
```

The stack

<table>
<thead>
<tr>
<th>main</th>
<th>a</th>
</tr>
</thead>
</table>

The heap

| main | i |
The heap

- Example execution

```cpp
int main() {

    // i points to an int in the heap
    int* i = new int;

    // affects 65 to *i
    *i = 65;

    // a points to an array of chars in the heap
    char* a = new char[10];

    // affects 'A' to the 5th element of the array
    a[4] = 'A';

    // free the memory pointed by i
    delete i;

    // free the memory pointed by a
    delete[] a;

    return 0;
}
```

The stack

The heap
int main() {

    // i points to an int in the heap
    int* i = new int;

    // affects 65 to *i
    *i = 65;

    // a points to an array of chars in the heap
    char* a = new char[10];

    // affects 'A' to the 5th element of the array
    a[4] = 'A';

    // free the memory pointed by i
    delete i;

    // free the memory pointed by a
    delete[] a;

    return 0;
}
Classes
Classes

- Motivation: organize data and algorithms in large programs

- **Objects** are data structures that may contain
  - Data members
  - Member functions

- The word “Class” is used to mean “Class of objects”
  - All objects of the same class have the same structure
  - Only **one class** with a given name, but **many objects** of the same class

- **First**, you define a class of objects, **then** you create objects of this class

- Fancy sentences:
  - “Object a is an **instance** of class A” (note the (arbitrary) capitalization)
  - “The **type of** “a” is “A””
The minimal class

The class name

The class keyword

```
class A {
};
```

There has to be a ;
A simple class

```cpp
class A {
public:
    int i;
    void print() {
        std::cout << i << std::endl;
    }
};
```
Classes

- Declaring a class means **making a new type** (like `int`, `float`, etc.)

```cpp
int i;  // The type of i is int
A a;   // The type of a is A
```

The type name     The variable name
Data members
Data members

```cpp
class A {

public:

    int i;

    void print() {
        std::cout << i << std::endl;
    }

};
```

- If a class has data members, then objects of this class **each have their own instances of these variables**

```cpp
A a; // a has its own int i
A b; // b has its own int i
A c; // c has its own int i
```

- In other words, an object "contains" data
Data members

```c
A a;    // a has its own int i
A b;    // b has its own int i
A c;    // c has its own int i
```

Some memory has been allocated for each object and its variables

**Question**: which type of memory has been used in this example?
Data members

A a; // a has its own int i
A b; // b has its own int i
A c; // c has its own int i

Answer: these objects have been allocated on the stack (since new hasn’t been used) (more on this later)
Data members

• From **outside**, data members are accessed with `.`

What about here, see the dot?

```cpp
a.i=10; // assigns 10 to i of a
std::cout << a.i << std::endl; // outputs 10
```

Look Ma’, there’s a dot!
Data members

• Data members **may be objects** (this is called **composition**)

```cpp
class A {
  public:

    int i;

};

class B {
  public:

    A a;

};

int main() {

  B b;
  b.a.i=10;  // assigns 10 to i of A
  std::cout << b.a.i << std::endl;  // outputs 10

  return 0;
}
```
Data members

- An object cannot contain itself (more on this later)

```cpp
class A {
    public:
        A a;
};
```
Compilation error
Member functions
Member functions

- Member functions **have access to data members**

```cpp
class A {

public:

    int i;

    void print() {
        std::cout << i << std::endl;
    }

};
```

This i is the one belonging to the objects
Member functions

• From **outside**, data members and **member functions** are accessed with **.**

```cpp
a.i=10;  // assigns 10 to i of a
a.print();  // outputs 10
```

*a.print() has access to a.i*
Member functions

- From **outside**, data members and **member functions** are accessed with .

```cpp
A a;
A b;

a.i=10; // assigns 10 to i of a
b.i=17; // assigns 17 to i of b

a.print(); // outputs 10
b.print(); // outputs 17
```
Member functions

• Example execution

```cpp
class A {

public:

    int i;

    void print() {
        std::cout << i << std::endl;
    }

};

int main() {

    A a;
    a.i=10; // assigns 10 to i of a
    a.print(); // outputs 10

    return 0;
}
```

The stack
Member functions

- Example execution

```cpp
class A {
  public:
    int i;
    void print() {
      std::cout << i << std::endl;
    }
  
};

int main() {
  A a;
  a.i=10; // assigns 10 to i of a
  a.print(); // outputs 10
  return 0;
}
```

The stack
**Member functions**

- **Example execution**

```cpp
class A {
    public:
        int i;
        void print() {
            std::cout << i << std::endl;
        }
};

int main() {
    A a;
    a.i=10; // assigns 10 to i of a
    a.print(); // outputs 10
    return 0;
}
```

The stack
Member functions

- Example execution

```cpp
class A {
    public:
        int i;
        void print() {
            std::cout << i << std::endl;
        }
    }

    int main() {
        A a;
        a.i = 10; // assigns 10 to i of a
        a.print(); // outputs 10
        return 0;
    }
}
```

The stack
Member functions

• Example execution

```cpp
class A {
public:
    int i;
    void print() {
        std::cout << i << std::endl;
    }
};

int main() {
    A a;
    a.i=10; // assigns 10 to i of a
    a.print(); // outputs 10
    return 0;
}
```

The stack
Member functions

• Example execution

```cpp
class A {
    public:
        int i;
        void print() {
            std::cout << i << std::endl;
        }
};

int main() {
    A a;
    a.i=10; // assigns 10 to i of a
    a.print(); // outputs 10
    return 0;
}
```

Since we called `print()` of `a`, this is `i` of `a`.

The stack

```
main  a  i 10
```
Member functions

• Example execution

```cpp
class A {
  public:
    int i;
    void print() {
      std::cout << i << std::endl;
    }
};

int main() {
  A a;
  a.i=10; // assigns 10 to i of a
  a.print(); // outputs 10
  return 0;
}
```

The stack

```
main    a
        ↓
        i
  10
```

The stack
Member functions

- Example execution

```cpp
class A {
    public:
        int i;
        void print() {
            std::cout << i << std::endl;
        }
    }

    int main() {
        A a;
        a.i=10; // assigns 10 to i of a
        a.print(); // outputs 10
        return 0;
    }
```
Member functions

- Example execution

```cpp
class A {

public:

    int i;

    void print() {
        std::cout << i << std::endl;
    }

};

int main() {

    A a;
    a.i=10; // assigns 10 to i of a
    a.print(); // outputs 10

    return 0;

}
```

The stack
Member functions

- Member functions may also be used to **modify** data members.

```cpp
class A {

public:

    int i;

    void add(int j) { i=i+j; }
    void print() { std::cout << i << std::endl; }

};

int main() {

    A a;
    a.i=10;       // assigns 10 to i of a
    a.add(4);    // adds 4 to i of a
    a.print();   // outputs 14

    return 0;

}
```
Member functions

- Member functions may be **overloaded**

```cpp
class A {

public:

    int i;

    void add(int j) { i=i+j; }
    void add(int j, int k) { i=i+j+k; }
    void print() { std::cout << i << std::endl; }

};

int main() {

    A a;
    a.i=10; // assigns 10 to i of a
    a.add(4); // adds 4 to i of a
    a.add(5,6); // adds 11 to i of a
    a.print(); // outputs 25

    return 0;

}
Access modifiers
• Access modifiers are used to specify **who** can access data and functions
  - **public**: anyone outside the class has access
  - **private**: nobody outside the class has access
  - (more on a **third modifier** later)

• Modifiers may be used in **zero or more places**
  - **If no modifier is specified, everything is private**

class A {
    public:
        int i;
};

int main() {
    A a;
    a.i = 10;  // assigns 10 to i of A
    std::cout << a.i << std::endl;  // outputs 10
    return 0;
}
Access modifiers

```cpp
class A {
private:
    int i;
};

int main() {
    A a;
    a.i = 10;                   // assigns 10 to i of A
    std::cout << a.i << std::endl;  // outputs 10

    return 0;
}
```

Everything below `private` is inaccessible from outside.
Access modifiers

```cpp
class A {
    public:
    void setI(int a) { i=a; }
    int getI() { return i; }

    private:
    int i;

};

int main() {

    A a;
    a.setI(10);  // assigns 10 to i of A
    std::cout << a.getI() << std::endl;  // outputs 10

    return 0;

}
```

A public interface is used to access data. Implementation details are hidden from clients (i.e. users).
This
• In each class, there is a **reserved** (hidden but usable) variable called **this**
• If the class name is A, the type of **this** is A* (pointer to an object of the class)
• The (constant) value of **this** is the **address of the object**
• In each class, there is a reserved (hidden but usable) variable called \texttt{this}

• If the class name is \texttt{A}, the type of \texttt{this} is \texttt{A*} (pointer to an object of the class)

• The (constant) value of \texttt{this} is the address of the object

```cpp
class A {
  public:
    void setI(int a) { i=a; }
    int getI() { return i; }

  private:
    int i;
};

int main() {
  A a;
  a.setI(10);
  std::cout << a.getI() << std::endl;

  return 0;
}
```
• In each class, there is a reserved (hidden but usable) variable called this
• If the class name is A, the type of this is A* (pointer to an object of the class)
• The (constant) value of this is the address of the object

```cpp
class A {
    public:
        void setI(int a) { i=a; }
        int getI() { return i; }

    private:
        int i;
    
};

int main() {
    A* a=new A();
    a->setI(10);
    std::cout << a->getI() << std::endl;
    
    return 0;
}
```

(address 170875)

<table>
<thead>
<tr>
<th>this</th>
</tr>
</thead>
<tbody>
<tr>
<td>170875</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
• **this** may be used when there is an ambiguity between a data member and a member function argument

```cpp
class A {
    public:
        void setI(int i) { i=i; }
        int getI() { return i; }

    private:
        int i;
};
```
• **this** may be used when there is an ambiguity between a data member and a member function argument

```cpp
class A {
    public:
        void setI(int i) { i = i; }
        int getI() { return i; }

    private:
        int i;

};
```

If we forget **this**, the `i` of the object is not changed! (the argument has priority)
• **this** may be used when there is an ambiguity between a data member and a member function argument

```cpp
class A {
public:

    void setI(int i) { i = i; }
    int getI() { return i; }

private:

    int i;

};
```

If we forget **this**, the i of the object is not changed! (the argument has priority)

Here, **this** is not necessary
• **this** may be used when there is an ambiguity between a data member and a member function argument

```cpp
class A {
public:
    void setI(int i) { (*this).i = i; }
    int getI() { return i; }

private:
    int i;
};
```

**this** is used as a regular pointer

Here, **this** is not necessary
• **this** may be used when there is an ambiguity between a data member and a member function argument

```cpp
class A {
public:

    void setI(int i) { this->i = i; }
    int getI() { return i; }

private:

    int i;

};
```

*This* is used as a regular pointer

Here, *this* is not necessary
Constructors and destructors
Constructors

• Constructors are **special member functions**
  • They have the **same name as the class**
  • They have **no return type** (not even void)

• Constructors are used to initialize (“construct”) objects

```cpp
class A {
    public:
        A() { // constructor
            i=0;
        }
        int i;
};
```
Constructors

- Constructors are called automatically the first time objects are declared.

```cpp
A a; // The constructor A() is called
```

- As other functions, constructors may have parameters.

```cpp
class A {

public:

    A() { i=0; } // default constructor
    A(int a) { i=a; } // ninja constructor

    int i;

};
```
Constructors

- Using non-default constructors may be used to simplify code

```cpp
// using the default constructor

A a;     // The constructor A() is called
a.i = 10;

// using the non-default constructor

A a(10); // The constructor A(int a) is called
```

- The compiler automatically determines which constructor to call, based on the arguments types
Constructors

- If you program **no** constructors, the compiler makes a default one for you
  - (no guarantee on what it does though)

```cpp
A a;  // calls the implicit default constructor
std::cout << a.i << std::endl;  // outputs a random variable
```

On some compilers, this doesn’t even compile
(the compiler complains that ‘a’ has not been initialized properly)
Destructors

- Destructors are special member functions
  - They have the same name as the class, preceded with ~ (tilde)
  - They have no return type

- Destructors are called just before the object memory is released

- Destructors are typically used to clean up memory

```cpp
class A {

public:

    int* i;

    A() { i=new int[10]; }
    ~A() { delete [] i; }

};
```
Destructors

```cpp
class A {

public:

    int* i;

    A() { i=new int[10]; }  
    ~A() { delete [] i; } 
}

int main() {

    A a; // A() is called  

    for (int i=0;i<10;i++)
        std::cout << a.i[i] << std::endl;

    return 0;

} // ~A() is called
```
Destructors

```cpp
class A {

public:

    int* i;

    A() { i = new int[10]; }
    ~A() { delete [] i; }

};

int main() {

    A a; // A() is called

    for (int i=0; i<10; i++)
        std::cout << a.i[i] << std::endl;

    return 0;

} // ~A() is called
```

What is created on the stack?
What is created on the heap?
Destructors

```cpp
class A {
public:
    int* i;

    A() { i=new int[10]; }
    ~A() { delete [] i; }
};
```

The i of a (a.i) is on the stack (since a is on the stack)

The array is on the heap! (since we use `new` (even though i of a, on the stack, contains its address)

```cpp
int main() {
    A a; // A() is called

    for (int i=0;i<10;i++)
        std::cout << a.i[i] << std::endl;

    return 0;
}
```

a is on the stack

i is on the stack
Destructors

- There can be at most one destructor per class

- A destructor cannot take any arguments

- If no destructor is programmed, one is automatically written for you

- Beware of memory leaks (= memory that cannot be deleted anymore)
Organizing files – Part 1
Organizing files

- For large projects, **class declarations** are separated from **class definitions**
- One **header file** (e.g. `A.hpp`) and one **source file** (e.g. `A.cpp`) per class (e.g. `A`)

```cpp
#pragma once

class A {
public:
    A();
    ~A();
    int getI();
    void setI(int i);

private:
    int i;
};
```

So that the compiler includes this file at most once.
Organizing files

- For large projects, **class declarations** are separated from **class definitions**
- One **header file** (e.g. A.hpp) and one **source file** (e.g. A.cpp) per class (e.g. A)

Source file: A.cpp

```cpp
#include "A.hpp"

A::A() {
    i=0;
}

A::~A() {
    i=0;
}

int A::getI() { return i; }
void A::setI(int i) { this->i=i; }
```

**Member function definition syntax:**
```
returnType className::functionName(arguments) {
    statements...
}
```
Organizing files

- For large projects, **class declarations** are separated from **class definitions**
- One **header file** (e.g. A.hpp) and one **source file** (e.g. A.cpp) per class (e.g. A)

```
#pragma once

#include "A.hpp"

class B {
public:
    B();
    ~B();
    A getA();
    void setA(A a);

private:
    A a;
};
```

Include A.hpp so the compiler knows what A means
Organizing files

- For large projects, **class declarations** are separated from **class definitions**
- One **header file** (e.g. A.hpp) and one **source file** (e.g. A.cpp) per class (e.g. A)

Include B.hpp so the compiler knows what B means

**Source file: B.cpp**

```cpp
#include "B.hpp"

A B::getA() { return a; }
void B::setA(A a) { this->a = a; }
```

**Member function definition syntax:**
returnType className::functionName(arguments) {
  statements...
}

⚠️ It is not necessary to include A.hpp (it is included by B.hpp)
Organizing files

- For large projects, **class declarations** are separated from **class definitions**
- One **header file** (e.g. A.hpp) and one **source file** (e.g. A.cpp) per class (e.g. A)

```cpp
#include "B.hpp"
#include <iostream>

void main() {

    A a;
    a.setI(10); // a.i = 10

    B b;
    b.setA(a); // b.a = a

    std::cout << b.getA().getI() << std::endl; // outputs 10
}
```

It is not necessary to include A.hpp (it is included by B.hpp)
Static members
Static data members

- Static data members do **not** depend on a specific object
- Only **one version** of a static data member exists in memory
- Static data members are **shared** among all objects

```cpp
class A {
public:
    static int i;
};
```

- The type of the static variable
- The name of the class
- **::** is the **scope operator** (indicates where i is declared)
- i of A has to be defined outside the class (so that it exists somewhere in memory)
Static data members

class A {

public:

    static int i;
    void print() { std::cout << i << std::endl; }

};

int A::i;

int main() {

    A a;
    A b;

    a.i=10; // assigns 10 to i of A
    b.print(); // outputs 10

    return 0;

}
Static member functions

- Static member functions do **not** depend on a specific object

- Static member functions **may only access static data members** (since they are common to all objects, they are not part of any specific object)

```cpp
class A {
    public:
        int i;
        static void print() { std::cout << i << std::endl; }
};
```

Error, i is not static
Static member functions

• Static member functions do **not** depend on a specific object

• Static member functions **may only access static data members**
  (since they are common to all objects, they are not part of any **specific** object)

```cpp
class A {

public:

    static int i;

    static void print() { std::cout << i << std::endl; }

};
```

OK, i is static
Creating arrays of objects
Creating arrays of objects

- As with fundamental types, arrays can be created

```cpp
A a[10]; // calls the default constructor 10 times
a[4].i=17; // assigns 17 to the 5th object in the array
(a[4]).i=17; // assigns 17 to the 5th object in the array
```

The dot is used to access members

The parentheses are optional
Creating objects on the heap
Creating objects on the heap

- Like fundamental variables or arrays, objects may be created on the heap

```cpp
class A {
public:
    void setI(int a) { i=a; }
    int getI() { return i; }

private:
    int i;
};

int main() {
    A* a=new A(); // a is a pointer
    (*a).setI(10); // assigns 10 to i of A
    std::cout << (*a).getI() << std::endl; // outputs 10
    return 0;
}
```
Creating objects on the heap

class A {
public:

    void setI(int a) { i=a; }
    int getI() { return i; }

private:

    int i;
};

int main() {

    A* a=new A();
    (*a).setI(10);
    std::cout << (*a).getI() << std::endl;
    return 0;
}
Creating objects on the heap

```cpp
class A {
public:

    void setI(int a) { i=a; }
    int getI() { return i; }

private:

    int i;
};

int main() {

    A* a=new A();
    (*a).setI(10);
    std::cout << (*a).getI() << std::endl;
    return 0;
}
```

The stack

The heap
Creating objects on the heap

class A {
public:
    void setI(int a) { i=a; }
    int getI() { return i; }

private:
    int i;
};

int main() {
    A* a=new A();
    (*a).setI(10);
    std::cout << (*a).getI() << std::endl;
    return 0;
}
Creating objects on the heap

class A {
public:

    void setI(int a) { i=a; }
    int getI() { return i; }

private:

    int i;
};

int main() {

    A* a= new A();
    (*a).setI(10);
    std::cout << (*a).getI() << std::endl;
    return 0;
}
Creating objects on the heap

Like fundamental variables or arrays, objects **may be created on the heap**

```cpp
class A {
    public:

    void setI(int a) { i=a; }
    int getI() { return i; }

    private:

    int i;

};

int main() {
    A* a=new A();  // a is a pointer
    (*a).setI(10); // assigns 10 to i of A
    std::cout << (*a).getI() << std::endl; // outputs 10
    return 0;
}
```

Dereferencing operator
Creating objects on the heap

- Like fundamental variables or arrays, objects may be created on the heap.

```cpp
class A {
public:

    void setI(int a) { i=a; }
    int getI() { return i; }

private:

    int i;

};

int main() {

    A* a=new A();                 // a is a pointer
    *a.setI(10);                  // assigns 10 to i of A
    std::cout << *a.getI() << std::endl; // outputs 10

    return 0;

}
```

Parentheses are necessary (* has low priority)
Creating objects on the heap

- Like fundamental variables or arrays, objects may be created on the heap

```cpp
class A {
public:

    void setI(int a) { i=a; }
    int    getI() { return i; }

private:

    int i;

};

int main() {

    A* a=new A(); // a is a pointer
    a->setI(10);  // assigns 10 to i of A
    std::cout << a->getI() << std::endl; // outputs 10

    return 0;

};
```

This is an arrow that points to the content of *a
Creating objects on the heap

- Like fundamental variables or arrays, objects **may be created on the heap**

```cpp
class A {
public:

    void setI(int a) { i=a; }
    int getI() { return i; }

private:

    int i;

};

int main() {

    A* a = new A(); // a is a pointer
    a->setI(10); // assigns 10 to i of A
    std::cout << a->getI() << std::endl; // outputs 10

    return 0;

}
```

The variable `a` **points to an object of class A**
Argument passing
Argument passing

- Objects may be passed to functions (member or not) by value or by reference.

- When an object is passed by value, a copy of it is made (might be slow).

```cpp
class A {
    public:
        A(int i) { this->i = i; }
        int i;
    }

    void increment(A o) {
        o.i++;
    }

int main() {
    A a(10); // a.i=10
    increment(a); // does nothing to a
    std::cout << a.i << std::endl; // outputs 10
    return 0;
}
```

- o is a copy of the passed argument.
- o is destroyed when we exit increment.
Argument passing

- Objects may be passed to functions (member or not) **by value** or **by reference**

- When an object is **passed by value**, a copy of it is made (might be slow)

```cpp
class A {
public:
    A(int i) { this->i=i; }  // this is a copy
    int i;
};

void increment(A a) {
    a.i++;
}

int main() {
    A a(10);                // a.i=10
    increment(a);           // does nothing to a
    std::cout << a.i << std::endl; // outputs 10
    return 0;
}
```

Even if the argument `a` has the same name, it is **still a copy**

The copy `a` is **destroyed** when we exit `increment`
Argument passing

- Objects may be passed to functions (member or not) by **value** or by **reference**

- When an object is **passed by value**, a copy of it is made (might be slow)

```cpp
class A {
    public:
        A(int i) { this->i=i; }
        int i;
};

void increment(A& r) {
    r.i++; // 'r' is a reference to (another name for) the passed argument
}

int main() {
    A a(10); // a.i=10
    increment(a); // increments a.i
    std::cout << a.i << std::endl; // outputs 11
    return 0;
}

This modifies 'a'

The reference 'r' is **forgotten** when we exit increment
```
Overloading operators
Overloading operators

class A {
  public:

    A(int i) { this->i=i; }
    A plus(A a) { return A(i+a.i); }
    int getI() { return i; }

  private:
    int i;
  }

  int main() {

    A a(10);       // a.i=10
    A b(10);       // b.i=10
    A c=a.plus(b); // c.i=a.i+b.i
    std::cout << c.getI() << std::endl; // outputs 20
    return 0;

  }

Overloading operators

class A {
    public:
        A(int i) { this->i=i; }
        A plus(A a) { return A(i+a.i); }
        int getI() { return i; }

    private:
        int i;
    
};

int main() {
    A a(10); // a.i=10
    A b(10); // b.i=10
    A c=a.plus(b); // c.i=a.i+b.i
    std::cout << c.getI() << std::endl; // outputs 20
    return 0;
}

getI() is not necessary here! ('plus' and 'i' belong to the same class)
Overloading operators

class A {
    public:

    A(int i) { this->i=i; }
    A plus(A a) { return A(i+a.i); }
    int getI() { return i; }

    private:
    int i;
};

int main() {

    A a(10); // a.i=10
    A b(10); // b.i=10
    A c=a.plus(b); // c.i=a.i+b.i
    std::cout << c.getI() << std::endl; // outputs 20
    return 0;

}
Overloading operators

class A {
public:

  A(int i) { this->i=i; }  
  A operator+(A a) { return A(i+a.i); }
  int getI() { return i; }

private:
  int i;
};

int main() {

  A a(10);    // a.i=10
  A b(10);    // b.i=10
  A c=a+b;    // c.i=a.i+b.i
  std::cout << c.getI() << std::endl; // outputs 20
  return 0;

}
Overloading operators

```cpp
class A {
    public:
        A(int i) { this->i = i; }
        A operator+(A& a) { return A(i+a.i); }
        int getI() { return i; }

    private:
        int i;
    }

    int main() {
        A a(10);              // a.i=10
        A b(10);              // b.i=10
        A c = a + b;          // c.i=a.i+b.i
        std::cout << c.getI() << std::endl; // outputs 20
        return 0;
    }
```
Inheritance
Inheritance is a C++ mechanism that makes it possible to reuse functionality.

Inheritance should mean ‘is a’

A is a **base class** of B

```cpp
class A {
    public:
        int i;
    }
}
```

B inherits A (everything public in A is public in B)

B is a **derived class** of A

```cpp
class B : public A {
    public:
        int j;
    }
}
```

An object of class B is an object of class A, with something more

```cpp
int main() {
    B b;
    b.i=10; // assigns 10 to b.i
    b.j=20; // assigns 20 to b.j
    return 0;
}
```
Inheritance

- Inheritance is a C++ mechanism that makes it possible to reuse functionality

- Inheritance **should mean ‘is a’**

```cpp
class A {
    public:
        int i;
        int getI() { return i; }
};

class B : public A {
    public:
        int j;
        int getJ() { return j; }
};

int main() {
    B b;
    b.i=10;  // assigns 10 to b.i
    b.j=20;  // assigns 20 to b.j
    std::cout << b.getI() << std::endl;  // outputs 10
    return 0;
}
```

Public functions of A are public in B
Inheritance

- Inheritance is a C++ mechanism that makes it possible to reuse functionality

- Inheritance **should** mean ‘is a’

```cpp
class A {
public:
    int getI() { return i; }
private:
    int i;
};

class B : public A {
public:
    int j;
    void print() { std::cout << i << std::endl; }
};
```

i is private in A, so this does not compile
Inheritance

- Inheritance is a C++ mechanism that makes it possible to reuse functionality.

- Inheritance should mean ‘is a’.

```cpp
class A {
    public:
        int getI() { return i; }
    private:
        int i;
};

class B : public A {
    public:
        int j;
        void print() { std::cout << getI() << std::endl; }
};
```

getI() is public in A, so this compiles.
Inheritance

- Inheritance is a C++ mechanism that makes it possible to reuse functionality
- Inheritance should mean ‘is a’

```cpp
class A {
public:
    int getI() { return i; }
protected:
    int i;
};

class B : public A {
public:
    int j;
    void print() { std::cout << i << std::endl; }
};
```

i is **protected**, so is accessible through **public inheritance**
Inheritance

• When constructing a derived object, a base constructor is called first
• When destructing a derived object, the base destructor is called after

```cpp
class A {
public:
    A() { i=0; }
    ~A() {}
protected:
    int i;
};

class B : public A {
public:
    B() { j=0; }
    ~B() {}
protected:
    int j;
};

int main() {
    B b;  // calls A(), then B()
    return 0;
}  // calls ~B(), then ~A()
```
Virtual functions
Virtual functions

• A function that is redefined in a derived class **hides** the base class function

```cpp
class A {
public:
    void print() { std::cout << "A" << std::endl; }
};

class B : public A {
public:
    void print() { std::cout << "B" << std::endl; }
};

int main() {
    B b;
    b.print(); // outputs B
    return 0;
}
```
Virtual functions

- A function that is redefined in a derived class **hides** the base class function

```cpp
class A {
public:
    void print() { std::cout << "A" << std::endl; }
};

class B : public A {
public:
    void print() { std::cout << "B" << std::endl; }
};

int main() {
    A* b = new B();
    b->print(); // outputs A
    return 0;
}
```
Virtual functions

- The code of a virtual function is **located at runtime**, based on the object type (and not based on the variable type)

```cpp
class A {
public:
    virtual void print() { std::cout << "A" << std::endl; }
};

class B : public A {
public:
    virtual void print() { std::cout << "B" << std::endl; }
};

int main() {
    A* b = new B();
    b->print(); // outputs B
    return 0;
}
```
Virtual functions

- The code of a virtual function is **located at runtime**, based on the object type (and not based on the variable type)

```cpp
class A {
public:
    virtual void print() { std::cout << "A" << std::endl; }
};

class B : public A {
public:
    virtual void print() { std::cout << "B" << std::endl; }
};

int main() {
    A* b = new B();
    b->print(); // outputs B
    return 0;
}
```
Organizing files – Part 2
Organizing files

- For large projects, **class declarations** are separated from **class definitions**

```cpp
#pragma once

class A {
public:
    A();
    ~A();
    int getI();
    void setI(int i);

protected:
    int i;
};
```

So that the compiler includes this file at most once
Organizing files

- For large projects, **class declarations** are separated from **class definitions**

  **Syntax**: returnType className::functionName(arguments) { statements }

```cpp
#include "A.hpp"

A::A() {
    i=0;
}

A::~A() {
    i=0;
}

int A::getI() { return i; }
void A::setI(int i) { this->i=i; }
```
# Organizing files

- For large projects, **class declarations** are separated from **class definitions**

```cpp
#include "A.hpp"

class B : public A {
public:
    int getJ();
    void setJ(int j);

protected:
    int j;
};
```

So that the compiler includes this file at most once.
Organizing files

- For large projects, **class declarations** are separated from **class definitions**

```
#include "B.hpp"

int B::getJ() { return j; }
void B::setJ(int j) { this->j = j; }
```
Organizing files

- For large projects, **class declarations** are separated from **class definitions**

```cpp
#include "A.hpp"
#include "B.hpp"

#include <iostream>

int main() {
    A* b = new B();
    b->setI(10); // b.i=10
    std::cout << b->getI() << std::endl; // outputs 10
    return 0;
}
```
Organizing files

- To prevent `#include` cycles, **forward declarations** might be necessary

```cpp
#pragma once

class B;

class A {
public:
    A();
    ~A();
    int getI();
    void setI(int i);

protected:
    int i;
    B* b;
};
```

Forward declaration (tells the compiler that B is a valid symbol)
Inheritance (Ninja)
Inheritance (Ninja)

```cpp
class A {
public:
    int i;
};

class B : public A {
    public:
        int j;
};

int main() {
    B b;
    b.i=10;  // assigns 10 to b.i
    b.j=20;  // assigns 20 to b.j

    return 0;
}
```

Public inheritance

Everything public in A is public in B
# Inheritance (Ninja)

<table>
<thead>
<tr>
<th>Base class modifier</th>
<th>Type of inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Public</td>
</tr>
<tr>
<td>Public</td>
<td>Public</td>
</tr>
<tr>
<td>Protected</td>
<td>Protected</td>
</tr>
<tr>
<td>Private</td>
<td>Hidden (not accessible)</td>
</tr>
</tbody>
</table>
Ninja tricks
Typedef

- The **typedef** keyword may be used to define types!

```cpp
class A {
public:
    int i;
};

typedef A B;  // B is just another name for the class (the type) A

int main() {
    B b;          // b is an object of class B=A
    b.i=10;       // assigns 10 to b.i
    std::cout << b.i << std::endl; // outputs 10
    return 0;
}
```
**Constness**

- The `const` keyword may be used to specify constant values.

```cpp
int main() {
    const int i = 17;
    i = 12;  // This does not compile: the value of i cannot be changed

    return 0;
}
```
**Constness**

- The `const` keyword may be used to **specify constant values**.

```cpp
int multiplyByTwo(const int i) {
    i *= 2;  // This does not compile: the value of i cannot be changed (the i of multiplyByTwo)
    return i;
}

int main() {
    int i = 12;
    i = multiplyByTwo(i);
    return 0;
}
```
• The `const` keyword may be used to specify constant values.

```c++
int multiplyByTwo(int& i) {
    return 2 * i;
}

int main() {
    int a = 17;
    int b = 8;
    int c = multiplyByTwo(a+b);
    return 0;
}
```

This does not compile: we cannot pass a reference to the non-const temporary value `a+b`
**Constness**

- The `const` keyword may be used to specify constant values.

```cpp
int multiplyByTwo(const int& i) {
    return 2 * i;
}
```

```cpp
int main() {
    int a = 17;
    int b = 8;
    int c = multiplyByTwo(a+b);

    return 0;
}
```

This compiles: even though `a+b` is a temporary value, the compiler knows we will not modify it, since the `i` of `multiplyByTwo` is a reference to a const value.
Constness

- The `const` keyword may be used to specify non-modifying functions.

```cpp
class A {

public:

    void setI(int i) { this->i = i; }
    int getI() const { return i; }

private:

    int i;

};
```

The function `getI` cannot change the value of `i` (this helps the compiler optimize code, and is useful information for class clients).
Namespaces

- Namespaces may be used to **organize code** and **avoid name conflicts**

```cpp
namespace MyNameSpace {

    class A {
        public:
            int i;
    };
}

int main() {

    MyNameSpace::A a; // a is an object of class MyNameSpace::A
    a.i=10; // assigns 10 to a.i
    std::cout << a.i << std::endl; // outputs 10

    return 0;
}
```

Namespace declaration

The scope operator `::` indicates to which namespace the class belongs.
Namespaces

• Namespaces may be used to **organize code** and **avoid name conflicts**

```cpp
namespace MyNameSpace {

    class A {
        public:
            int i;
    }

}

int main() {

    A a;
    a.i = 10;
    std::cout << a.i << std::endl;  // outputs 10

    return 0;

}
```

Namespace declaration

If the namespace is not specified, the program does not compile
Namespaces

- Namespaces may be used to **organize code** and **avoid name conflicts**

```c++
namespace MyNameSpace {

    class A {
        public:
            int i;
    }

}
```

The `using` keyword specifies namespaces that the programmer wants to use.

```c++
using namespace MyNameSpace;
```

A is found in the used namespace, so there is no compiler error.

```c++
int main() {

    A a;
    a.i=10;
    std::cout << a.i << std::endl; // outputs 10

    return 0;
}
```
Namespaces

• Namespaces may be defined in several parts

```cpp
#pragma once

namespace MyNameSpace {
    class A {
        public:
            int i;
    };
}
```

```cpp
#pragma once

#include "A.hpp"

namespace MyNameSpace {
    class B : public A {
        public:
            int j;
    };
}
```
#include <iostream>

#include "A.hpp"
#include "B.hpp"

using namespace MyNameSpace;

int main() {

    B b; // b is an object of class MyNameSpace::B
    b.i=10; // assigns 10 to b.i
    b.j=20; // assigns 20 to b.j

    return 0;
}

main.cpp
Namespaces

- Namespaces may be **nested**

```cpp
namespace MyNameSpace {
    namespace MySubNameSpace1 {
        class A {
            public:
                int i;
        }
    }

    namespace MySubNameSpace2 {
        class B {
            public:
                int i;
        }
    }
}
```
Namespaces

• Namespaces may be **nested**

```cpp
int main() {
    MyNameSpace::MySubNameSpace1::A a;  // a is an object of class MyNameSpace::MySubNameSpace1::A
    MyNameSpace::MySubNameSpace2::B b;  // b is an object of class MyNameSpace::MySubNameSpace2::B

    return 0;
}
```
Namespaces

- Namespaces may be nested

```cpp
using namespace MyNameSpace;

int main() {
    MySubNameSpace1::A a; // a is an object of class MyNameSpace::MySubNameSpace1::A
    MySubNameSpace2::B b; // b is an object of class MyNameSpace::MySubNameSpace2::B

    return 0;
}
```

The compiler looks for class names inside MyNameSpace

OK, MySubNameSpace1::A and MySubNameSpace2::B are defined in MyNameSpace
Namespaces

• Namespaces may be **nested**

```cpp
using namespace MyNameSpace::MySubNameSpace1;

int main() {
    A a; // a is an object of class MyNameSpace::MySubNameSpace1::A
    MySubNameSpace2::B b; // b is an object of class MyNameSpace::MySubNameSpace2::B
    return 0;
}
```

The compiler looks for class names inside `MyNameSpace::MySubNameSpace1`

The type `MySubNameSpace1::B` is not found inside `MyNameSpace::MySubNameSpace1`
Nested classes

- Classes may be **nested** (defined within other classes)

```cpp
class A {
public:
    class B {
        public:
            int i;
    }
    B b;
};

int main() {
    A a;           // a is an object of class A
    A::B b;        // a.b is an object of class A::B
    a.b.i=10;      // a.b.i is the i of the b in a
}
```
Enumerations

- The `enum` keyword is used to assign readable values to integers.

```c++
enum MyEnum {
    MyFirstValue=17,
    MySecondValue=8,
    MyThirdValue=75
};

int main() {
    MyEnum v=MyFirstValue; // v=17
    return 0;
}
```

There has to be a ;
Enumerations

- The `enum` keyword is used to assign **readable values to integers**

```cpp
class A {
public:
    enum MyEnum {
        MyFirstValue=17,
        MySecondValue=8,
        MyThirdValue=75
    }
};

int main() {
    A::MyEnum v=A::MyFirstValue;    // v=17
    return 0;
}
```
Function templates
Function templates

- Reminder: functions may be overloaded

```cpp
int max(int a, int b) {
    if (a>b) return a;
    return b;
}

double max(double a, double b) {
    if (a>b) return a;
    return b;
}

int main() {
    std::cout << max(1,2) << std::endl; // calls max(int,int) and outputs 2
    std::cout << max(1.5,2.5) << std::endl; // calls max(double,double) and outputs 2.5
    return 0;
}
```
Function templates

• Reminder: functions may be overloaded

```cpp
int max(int a, int b) {
    if (a > b) return a;
    return b;
}

double max(double a, double b) {
    if (a > b) return a;
    return b;
}

int main() {

    std::cout << max(1,2) << std::endl;  // calls max(int,int) and outputs 2
    std::cout << max(1.5,2.5) << std::endl; // calls max(double,double) and outputs 2.5

    return 0;
}
```

**Problem:** how do we avoid writing functions for all useful arguments?
Function templates

• Reminder: functions may be overloaded

```cpp
int max(int a, int b) {
    if (a > b) return a;
    return b;
}

double max(double a, double b) {
    if (a > b) return a;
    return b;
}

int main() {
    std::cout << max(1, 2) << std::endl;  // calls max(int,int) and outputs 2
    std::cout << max(1.5, 2.5) << std::endl;  // calls max(double,double) and outputs 2.5
    return 0;
}
```

Solution: use function templates
Function templates

The `template` keyword

An abstract `type name T`

```cpp
template <typename T> T max(T a, T b) {
    if (a > b) return a;
    return b;
}
```
Function templates

- The compiler **instantiates** specialized versions of the functions templates, i.e. generates specialized code **where necessary**

```cpp
template <typename T> T max(T a, T b) {
    if (a>b) return a;
    return b;
}

int main() {
    std::cout << max(1,2) << std::endl;       // calls max(int,int) and outputs 2
    std::cout << max(1.5,2.5) << std::endl;   // calls max(double,double) and outputs 2.5
    return 0;
}
```

In this example, the template is **instantiated twice** (once for int, once for double)
Function templates

- The compiler **instantiates** specialized versions of the functions templates, i.e. generates specialized code **where necessary**

```cpp
#include <iostream>

template <typename T> T max(T a, T b) {
    if (a > b) return a;
    return b;
}

int main() {
    std::cout << max<int>(1.5, 2.5) << std::endl; // calls max(int,int) and outputs 2
    return 0;
}
```

The specialization may be **forced** (here, the compiler generates a warning because of the conversion of the arguments)
Function templates

- If the template is not used, its code is not compiled

```cpp
#include <iostream>

template <typename T> void function(T t) {
    the compiler does not care
}

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

This program compiles just fine
Function templates

- If the template is not used, its code is not compiled.

```cpp
#include <iostream>

template <typename T> void function(T t) {
    the compiler finds 15 errors in the program
}

int main() {
    std::cout << "Hello World!" << std::endl;
    function(1.0);
    return 0;
}
```

The compiler attempts to compile this

The compiler instantiates ‘function’ and tries to compile the instantiated version.
Function templates

The `template` keyword

An abstract `type name` T

```cpp
template <typename T> T max(T a, T b) {
    if (a > b) return a;
    return b;
}
```

In this example, the programmer assumes that the template will be used with types for which operator `>` is well defined.
Function templates

class A {
public:
    A(int i) { this->i=i; }  // A does not define operator >
    int i;
};

int main() {

    int i=max(1,2);         // calls max(int,int) and assigns 2 to i
    double d=max(1.5,2.5);  // calls max(double,double) and assigns 2.5 to d
    A a(1);
    A b(2);
    A c=max(a,b);           // does not compile ("A does not define >")

    return 0;
}
Function templates

class A {
    public:
        A(int i) { this->i=i; }
        int i;
        bool operator>(const A& a) { return (i>a.i); }
};

int main() {
    int i=max(1,2);    // calls max(int,int) and assigns 2 to i
    double d=max(1.5,2.5); // calls max(double,double) and assigns 2.5 to d

    A a(1);
    A b(2);
    A c=max(a,b);       // calls max(A,A) and assigns 2 to c.i

    return 0;
}
Function templates

• Function templates may have several template parameters

• Structure

template <template parameters> returnType functionName(arguments) {
    statements
}

These may depend on template parameters
Function templates

- Function templates may have several template parameters

```cpp
template <typename T1, typename T2> void print(T1 a, T2 b) {
    std::cout << a << " " << b << std::endl;
}

int main() {
    print(1, 2.0);  // calls print(int,double) and outputs 1 2
    print("The answer is", 42);  // calls print(char*,int) and outputs The answer is 42

    while(1);
    return 0;
}
```
Function templates

- Function templates may be **explicitly specialized**

```cpp
template <typename T> void print(T t) {
    std::cout << "General case: " << t << std::endl;
}

template <> void print<int>(int t) {
    std::cout << "Special case for int: " << t << std::endl;
}

int main() {
    print(2.0); // calls print(double) and outputs General case: 2
    print(2);  // calls print(int) and outputs Special case for int: 2
    return 0;
}
```
Function templates

- If a regular function with matching signature exists, it has priority

```cpp
#include <iostream>

template <typename T> void print(T t) {
    std::cout << "Generic template version: " << t << std::endl;
}

template <> void print<int>(int t) {
    std::cout << "Specialized template version: " << t << std::endl;
}

void print(int i) {
    std::cout << "Regular version: " << i << std::endl;
}

int main() {
    print(1.0);   // outputs Generic template version: 1
    print(1);     // outputs Regular version: 1
    print<int>(1); // outputs Specialized template version: 1
}
```
Function templates

- Function template parameters may be **integers**

```cpp
#include <iostream>

template <int numberOfPrints, typename T> void print(T t) {
    for (int i=0; i<numberOfPrints; i++) std::cout << t << std::endl;
}

int main() {
    print<10,int>(5); // outputs 5 ten times
    return 0;
}
```
Function templates

- To be reusable, templates definitions are typically placed in **header files** (since the compiler needs the definition to instantiate the template)

```
#pragma once

#include <iostream>

template <typename T1, typename T2> void print(T1 a, T2 b) {
    std::cout << a << " " << b << std::endl;
}
```
Class templates
Class templates

• Like function templates, class templates are used to write **generic classes**

• Structure

```cpp
template <template parameters> class className {

    statements

};
```

These may depend on template parameters
Class templates

- Template classes are instantiated where necessary

```cpp
template <typename T> class A {
    public:
        T t;
};

int main() {
    A<int> a; // a.t is an int
    a.t=10;  // assigns 10 to a.t

    A<char*> b; // b.t is a char*
    b.t=new char[10]; // assigns to b.t the address of a char array

    return 0;
}
```

The template parameter is obligatory
Class templates

• Template parameters may be classes

```cpp
class A {
public:
    A(int i) { this->i=i; }
    int i;
};

template <typename T> class B {
public:
    B(T a) : t(a) {}
    T t;
};

int main() {
    B<int> b(A(10)); // b.t is an A, whose i is an int equal to 10
    std::cout << b.t.i << std::endl; // outputs 10
    return 0;
}
```
Class templates

- Template parameters **may be template classes**

```cpp
template <typename T> class A {
public:
    A(T a) : t(a) {}
    T t;
};

int main() {
    A<A<int> > a(A<int>(10)); // a.t is an A<int>, whose t is an int equal to 10
    std::cout << a.t.t << std::endl; // outputs 10
    return 0;
}
```
Class templates

• Template parameters may be template classes

```cpp
template <typename T> class A {
public:
    A(T a) : t(a) {}
    T t;
};

int main() {
    A<int> a(A<int>(10)); // a.t is an A<int>, whose t is an int equal to 10
    std::cout << a.t.t << std::endl; // outputs 10
    return 0;
}
```

Always add a space between two consecutive ‘>’ (some compilers read ‘operator>>’).
The standard template library
Where we show a few examples of template classes and algorithms
The standard template library (STL)

- The standard template library
  - **Standard**: provided
  - **Template**: customizable
  - **Library**: include appropriate header files to use

```cpp
#include <iostream>

#include <string>

int main() {
    std::string s = "Hello World!";  // s in a string
    std::cout << s << std::endl;    // outputs Hello World!

    return 0;
}
```
The standard template library (STL)

- The standard template library
  - **Standard**: provided
  - **Template**: customizable
  - **Library**: include appropriate header files to use

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

int main() {

    std::string s = "Hello World!";  // s in a string
    std::cout << s << std::endl;     // outputs Hello World!

    return 0;
}
```

The STL is in the namespace `std`

string is a template class (inside ‘xstring’)

```cpp
typedef basic_string<char, char_traits<char>, allocator<char>> string;
```
The standard template library (STL)

- The standard template library defines several useful data structures...
  - Lists: `std::list<T>`
  - Strings: `std::string<T>`
  - Vectors: `std::vector<T>`
  - ...
- ... functionals (e.g. objects that perform specific functions)
  - `greater<T>`
  - ...
- ... and algorithms
  - `for_each`
  - `replace`
  - `sort`
  - ...
std::list<T>

- std::list<T> implements a **doubly-linked list**
  - O(1) push and pop (**front and back**)
  - O(n) access inside

The header that declares the list template class

```cpp
#include <list>

int main() {

    std::list<int> l;  // l is an empty list of int
    l.push_back(20);   // l={20}
    l.push_back(30);   // l={20,30}
    l.push_front(10);  // l={10,20,30}
    l.pop_back();     // l={10,20}
    l.pop_front();    // l={10}

    return 0;

}
```
std::string

- **std::string** implements a **string of char**

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

int main() {

    std::string girl; // a girl has no name
    std::cin >> girl;
    std::cout << girl << std::endl;

    return 0;

}
```
std::vector<T>

- std::vector<T> implements an extensible array of elements
  - Amortized O(1) push and pop (back only)
  - O(1) access inside

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

int main() {

  std::vector<int> v; // v is an empty vector of int
  v.push_back(10); // v={10}
  v.push_back(20); // v={10,20}
  v.pop_back(); // v={10}
  v.push_back(30); // v={10,30}

  std::cout << v[0] << " " << v[1] << std::endl; // outputs 10 30

  return 0;
}
```

std::vector<T> implements operator[] to access vector elements
# Combining templates

```cpp
#include <list>
#include <string>
#include <vector>

int main() {

    std::string s; // s is an empty string
    std::vector< std::string > v; // v is an empty vector of strings
    std::list< std::vector< std::string > > l; // l is an empty list of vectors of strings

    s.push_back('a'); // s="a"
    v.push_back(s); // v={"a"}
    l.push_back(v); // l={{"a"}}
    l.push_back(v); // l={{"a"},{"a"}}

    return 0;
}
```

The STL template classes have ‘uniform’ designs

The `push_back` calls insert copies

Note:

- Include the necessary header files at the beginning of your code.
- Use templates for scalable and reusable code.
- Understand the types and the behavior of STL classes like `std::vector` and `std::list`.
- The `push_back` method inserts elements at the end of a container.
- Be aware of the copy insertion behavior when using `push_back` with template classes.
Iterators

- Iterators are objects that are used to access (iterate over) STL classes
- They are defined inside the classes themselves

```cpp
#include <list>
#include <iostream>

int main() {

    std::list<int> l;  // l is an empty list of int
    l.push_back(10);   // l={10}
    l.push_back(20);   // l={10,20}
    l.push_back(30);   // l={10,20,30}

    for (std::list<int>::iterator i=l.begin(); i!=l.end(); i++) {
        std::cout << (*i) << std::endl;
    }

    return 0;
}
```
Iterators

- Iterators are objects that are used to **access (iterate over) STL classes**
- They are defined inside the classes themselves

```cpp
#include <list>
#include <iostream>

int main() {
    std::list<int> l;       // l is an empty list of int
    l.push_back(10);        // l={10}
    l.push_back(20);        // l={10,20}
    l.push_back(30);        // l={10,20,30}

    for (std::list<int>::iterator i=l.begin(); i!=l.end(); i++) {
        std::cout << (*i) << std::endl;
    }

    return 0;
}
```

i is an **iterator**
Iterators

- Iterators are objects that are used to **access (iterate over)** STL classes
- They are defined inside the classes themselves

```cpp
#include <list>
#include <iostream>

int main() {

    std::list<int> l; // l is an empty list of int
    l.push_back(10);  // l={10}
    l.push_back(20);  // l={10,20}
    l.push_back(30);  // l={10,20,30}

    for (std::list<int>::iterator i=l.begin(); i!=l.end(); i++) {
        std::cout << (*i) << std::endl;
    }

    return 0;
}
```

`l.begin()` and `l.end()` are **iterators**
Iterators

• Iterators are objects that are used to access (iterate over) STL classes
• They are defined inside the classes themselves

```cpp
#include <list>
#include <iostream>

int main() {

    std::list<int> l; // l is an empty list of int
    l.push_back(10);  // l={10}
    l.push_back(20);  // l={10,20}
    l.push_back(30);  // l={10,20,30}

    for (std::list<int>::iterator i=l.begin(); i!=l.end(); i++) {
        std::cout << (*i) << std::endl;
    }

    return 0;
}
```

The iterator implements operator++
(which moves to the next element)
Iterators

- Iterators are objects that are used to access (iterate over) STL classes.
- They are defined inside the classes themselves.

```cpp
#include <list>
#include <iostream>

int main() {
    std::list<int> l; // l is an empty list of int
    l.push_back(10);  // l={10}
    l.push_back(20);  // l={10,20}
    l.push_back(30);  // l={10,20,30}

    for (std::list<int>::iterator i=l.begin(); i!=l.end(); i++) {
        std::cout << (*i) << std::endl;
    }

    return 0;
}
```

The iterator implements the dereferencing operator.
Iterators

- Iterators are objects that are used to **access (iterate over)** STL classes
- They are defined inside the classes themselves

```cpp
#include <list>
#include <iostream>

int main() {

    std::list<int> l;  // l is an empty list of int
    l.push_back(10);   // l={10}
    l.push_back(20);   // l={10,20}
    l.push_back(30);   // l={10,20,30}

    for (std::list<int>::iterator i=l.begin(); i!=l.end(); i++) {

        std::cout << (*i) << std::endl;
    }

    return 0;
}
```

The type of i is `std::list<int>::iterator`, so the type of *i is `int`
Iterators

- Iterators are objects that are used to **access (iterate over)** STL classes
- They are defined inside the classes themselves

```cpp
#include <list>
#include <iostream>

int main() {

    std::list<int> l; // l is an empty list of int
    l.push_back(10);  // l={10}
    l.push_back(20);  // l={10,20}
    l.push_back(30);  // l={10,20,30}

    for (std::list<int>::iterator i=l.begin(); i!=l.end(); i++) {
        std::cout << (*i) << std::endl;
    }

    return 0;
}
```

Think of iterators as **pointers that know how to move within the data structure**
std::greater<T>

- std::greater<T> is a **functional** (a structure which contains a function)

```cpp
#include <iostream>
#include <functional>

int main() {
    int i=10;   // this should be clear
    int j=20;   // this should be clear as well

    std::greater<int> g;    // g is a functional (a structure which contains a function)
    bool result=g(i,j);     // result=(i>j)

    return 0;
}
```
for_each

- **for_each** is a template algorithm used to apply operations to ranges

Structure

```cpp
template<typename IteratorType, typename FunctionType> FunctionType for_each(IteratorType first, IteratorType last, FunctionType function);
```
for_each

- **for_each** is a template algorithm used to apply operations to ranges.

```cpp
#include <list>
#include <algorithm>
#include <iostream>

class Printer {
    public:
        void operator()(int i) { std::cout << i << " "; }
    }

int main() {
    Printer p; // p is a Printer
    std::list<int> l; // l is an empty list of int
    l.push_back(10); // l={10}
    l.push_back(20); // l={10,20}
    l.push_back(30); // l={10,20,30}
    for_each(l.begin(), l.end(), p); // outputs 10 20 30

    return 0;
}
```
for_each

```cpp
#include <list>
#include <algorithm>
#include <iostream>
#include <functional>

template <typename T> class ComparisonPrinter {
public:
    ComparisonPrinter(T a) : t(a) {}
    void operator()(T i) { std::cout << std::greater<T>()(t, i) << " "; } 
    T t;
};

int main() {

    ComparisonPrinter<int> p(15); // p is a ComparisonPrinter<int> and p.t=15
    std::list<int> l; // l is an empty list of int
    l.push_back(10); // l={10}
    l.push_back(20); // l={10,20}
    l.push_back(30); // l={10,20,30}

    for_each(l.begin(), l.end(), p); // outputs 1 0 0

    return 0;
}
```
replace

- replace is a template algorithm used to replace elements in ranges

```cpp
#include <list>
#include <algorithm>

int main() {

    std::list<int> l; // l is an empty list of int
    l.push_back(10); // l={10}
    l.push_back(20); // l={10,20}
    l.push_back(10); // l={10,20,10}
    l.push_back(30); // l={10,20,10,30}
    replace(l.begin(), l.end(), 10, 15); // l={15,20,15,30}

    return 0;
}
```
sort

- sort is a template algorithm used to sort values in ranges

```cpp
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v; // v is an empty vector of int
    v.push_back(10);   // v={10}
    v.push_back(40);   // v={10,40}
    v.push_back(20);   // v={10,40,20}
    v.push_back(30);   // v={10,40,20,30}
    sort(v.begin(), v.end()); // v={10,20,30,40}

    return 0;
}
```
sort

- sort is a template algorithm used to sort values in ranges

```cpp
#include <vector>
#include <algorithm>

class A {
public:
    A(int i) { this->i=i; }
    int i;
};

bool compare(A a, A b) { return (a.i<b.i); }

int main() {
    std::vector<A> v; // v is an empty vector of A
    v.push_back(A(10)); // v={A(10)}
    v.push_back(A(30)); // v={A(10),A(30)}
    v.push_back(A(20)); // v={A(10),A(30),A(20)}
    v.push_back(A(40)); // v={A(10),A(30),A(20),A(40)}
    sort(v.begin(),v.end(),compare); // v={A(10),A(20),A(30),A(40)}

    return 0;
}
```
That’s all, folks!

- Congratulations!

- To apply what you’ve learned, head to the documentation center of the SAMSON platform for molecular modeling, create apps and distribute them on SAMSON Connect!

  https://documentation.samson-connect.net