INF585 – Programming C++
X2011 – First semester 2014

Session 1

Stephane Redon
NANO-D – INRIA
Welcome!

• Stephane Redon
  • Head of the NANO-D group at INRIA Grenoble – Rhône-Alpes
  • stephane.redon@inria.fr

• Romain Bompis
  • PhD student in CMAP – Polytechnique
  • bompis@cmap.polytechnique.fr
Welcome!

- The class (24 students) – self-assessment
Welcome!

• Practical introduction to C++ programming (from “noob” to “ninja”?)

• Structure
  • 4 sessions about programming (theory + exercises)
  • 5 sessions for collaborative development of a C++ project

• Important / difficult points are emphasized with

• Ask a question as soon as you have one!
Welcome!

• Today
  • Theory about the basics
    • Programs
    • Functions
    • Flow control
    • ...

• Exercises
Overview
Overview

• C++ was designed and implemented by Bjarne Stroustrup

• C++ has been used in many large-scale projects (from his page)
  • Autodesk - Maya (Visual effects)
  • Adobe - Photoshop (Image editing)
  • Apple - Finder
  • Bloomberg (Financial tools)
  • CERN - Tools (Data analysis)
  • D. E. Shaw - Tools (Financial analysis and trading software)
  • Dassault Systèmes – CATIA (Computer-Aided Design)
  • Google - Chromium (Operating System)
  • Microsoft - Windows, Office, Visual Studio, etc.
  • Mozilla – Firefox (Browser)
  • ...

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[]) {
    013113A0  push   ebp
    013113A1  mov    ebp,esp
    013113A3  sub    esp,0D8h
    013113A9  push   ebx
    013113AA  push   esi
    013113AB  push   edi
    013113AC  lea    edi,[ebp-0D8h]
    013113B2  mov    ecx,36h
    013113B7  mov    eax,0CCCCCCCCh
    013113BC  rep stos dword ptr es:[edi]

    // compute 10!
    int n=1;
    013113BE  mov    dword ptr [n],1

    for (int i=1;i<10;i++) n=n*i;
    013113C5  mov    dword ptr [ebp-14h],1
    013113CC  jmp    main+37h (01311307h)
    013113CE  mov    eax,dword ptr [ebp-14h]
    013113D1  add    eax,1
    013113D4  mov    dword ptr [ebp-14h],eax
    013113D7  cmp    dword ptr [ebp-14h],0Ah
    013113D8  jge    main+49h (013113E9h)
    013113DD  mov    eax,dword ptr [n]
    013113E0  imul   eax,dword ptr [ebp-14h]
    013113E4  mov    dword ptr [n],eax
    013113E7  jmp    main+2Eh (013113CEh)

    return n;
    013113E9  mov    eax,dword ptr [n]
}
```

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

• Memory can be modeled as a linear array

One byte (8 bits)

One bit (0 or 1)

• 8 bits = 1 byte
• 1024 bytes = 1 kilobytes = 1 KB
• 1024 KB = 1 megabytes = 1 MB
• 1024 MB = 1 gigabytes = 1 GB
• 1024 GB = 1 terabytes = 1 TB
• ...
Memory

• The memory cells have **addresses**

![Memory diagram](image)

- 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

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

The main function (has to be called main)

Return type (has to be int)

The function has to return a value
The traditional Hello world! program

```cpp
#include <iostream>

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

- **Input / output streams**
- **Include a header file**
  ```cpp
  #include <iostream>
  ```
- **A string**
  ```cpp
  std::cout << "Hello World!" << std::endl;
  ```
- **Standard output stream (more on this later)**
- **End line (more on this later)**
- **Overloaded operator (more on this later)**
- **Return 0**
  ```cpp
  return 0;
  ```
Comments
Comments

- Extremely important

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

- Think about your collaborators

- Think about your future you
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;
*/
Fundamental data types
Fundamental data types

- bool (one bit): true or false
  ```cpp
  bool b=true;
  ```
- char (one byte): integer between -128 and 127
  ```cpp
  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
  ```cpp
  short s=1203;
  ```
- int (usually four bytes): integer between -2,147,483,648 and 2,147,483,647
  ```cpp
  int i=0;
  ```
  ```cpp
  long l=12012930;
  ```
- char, int and long can be prefixed by ‘unsigned’
  ```cpp
  unsigned int i=4294967295;  // maximum unsigned int
  unsigned short s=-1;       // s=65535
  ```
Fundamental data types

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

  ```
  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)

  ```
  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

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

- C++ has rules for conversions between types

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

• Structure

    typeName variableName[size];

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

• Example

    ```
    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
```

Demo ‘Streams’
References and pointers
References and pointers

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

```cpp
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 0 1
```

Address ‘17 bytes’

Address ‘18 bytes’
References and pointers

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

  Note the &

  Syntax: `typeName& referenceName=variableName;`

  The reference has to be assigned immediately

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

```cpp
unsigned char i=165;  // The variable i is stored at address 17
unsigned char& r=i;  // r is just another name for i
```

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

Address ‘17 bytes’

Address ‘18 bytes’
A pointer is a variable which contains the address of another variable.

Declaration syntax: `typeName* pointerName;`

Note the `*`

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

The address of a variable is obtained with `&`.
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
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 the pointer `p`

  ```cpp
  *p = 65;
  std::cout << *p << std::endl;  // affects 65 to `i`
  std::cout << p << std::endl;   // prints `A`
  ```
• Variables are stored in memory, at a given **address**

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

The variable **p** points to the variable **i**.

Address ‘17 bytes’

Address ‘18 bytes’
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

The variable `p` is stored at address 23
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

The variable `p` is stored at address 23

- We can perform **pointer arithmetics**
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

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

Address ‘17 bytes’

Address ‘18 bytes’

```c
p++;
```
Variables are stored in memory, at a given address.

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

References and pointers

Address ‘17 bytes’

Address ‘18 bytes’

```cpp
p--;  // p points one position before i
```
• 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
```
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
```

```cpp
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.

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

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

- Address '17 bytes'
- Address '18 bytes'

```
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
```

*(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 facultative
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

// check whether i is equal to 2

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;
}
Tests

// check whether i is equal to 2

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

```
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

// 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++;
}
The for loop

// 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

```cpp
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

// 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

```
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.
This function returns nothing

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

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

This function receives one argument, a double called `d`.

This time, the function prints nothing, but it returns a double.
Functions

This function returns a double

This function receives one argument, a double called d

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

In this version, the unnecessary temporary variable has been removed
Functions

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

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

void printNumberMultipliedByTwo(double d) {
    double dMultipliedByTwo=multiplyByTwo(d);
    printNumber(dMultipliedByTwo);
}
```

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;
}
```

Second overloaded version

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

Arguments must differ
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.

```cpp
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++;  
}

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++;
}

int main() {
    int i = 65; // affects 65 to i
    increment(i); // pass i by value to increment
    std::cout << i << std::endl; // prints 66
    return 0;
}
```

r is a reference to the passed argument (note the &).
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;       // affects 65 to i
    increment(i);   // pass i by value to increment
    std::cout << i << std::endl;  // prints 66
    return 0;
}
```
The stack
The stack

- Second type of memory segment associated to an executable
- The stack is used by
  - 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 by
  • 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;
}
```

How many variables are stored in the stack?
The stack

• Second type of memory segment associated to an executable
• The stack is used by
  • variables declared in the function
  • arguments passed by value to functions

```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;
}
```

These variables are stored in 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

- 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>main</th>
<th>i</th>
<th>65</th>
</tr>
</thead>
</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
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>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>i</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>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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
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>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>
</table>

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

| main | i | 67 |
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>main</th>
<th>i</th>
<th>67</th>
</tr>
</thead>
</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

| main | i | 67 |
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 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

```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;
}
```
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;
}
```
```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 stack

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 stack

- 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 stack

The heap
Development
Development

• Typically
  • mkdir toto: make a directory called toto
  • cd toto: go inside the toto directory
  • (cd .. : go in the parent directory)
  • nedit main.cpp&
  • g++ main.cpp -o main
  • ./main

• For serious work, use professional tools, for example
  • Windows: Microsoft Visual C++ Express (free) / (Eclipse + MinGW)
  • Mac: XCode
  • Linux: Eclipse
Exercises

1. Compile and execute a minimal program, that does nothing.

2. Write a Hello World! Program.

3. Write a program that asks a user the number of letters in a word, then a word, and determines using an iterative algorithm whether the word is a palindrome.

4. Implement a recursive version of the palindrome test.

5. Implement the Sieve of Eratosthenes to find prime numbers between 1 and N, where N is entered by the user, using pointer arithmetics.

6. Implement an algorithm that produces an estimate of PI, by generating random points in a square and observing the ratio of points that fall into the inscribed circle.