CTI: School of Computer Science,
Telecommunications and Information Technology

CSC 309: **Object Oriented Programming in C++**

Massimo Di Pierro

Textbooks and Compiler

Course title:

Object Oriented Programming in C++

Instructor:

Prof. Massimo Di Pierro
(PhD in High Energy Theoretical Physics from Univ. of Southampton, UK)

Textbook:

Applications Programming in C++
Johnsonbaugh and Kalin, Prentice Hall

Optional reference book:

C++ in Plain English, 3/E
Overland, John Wiley & Sons

Suggested compiler and IDE:

Bloodshed Dev-C++ (mingw gcc) version 4
download from www.bloodshed.net/devcpp.html

Course web page:

http://www.cs.depaul.edu/courses/syllabus.asp

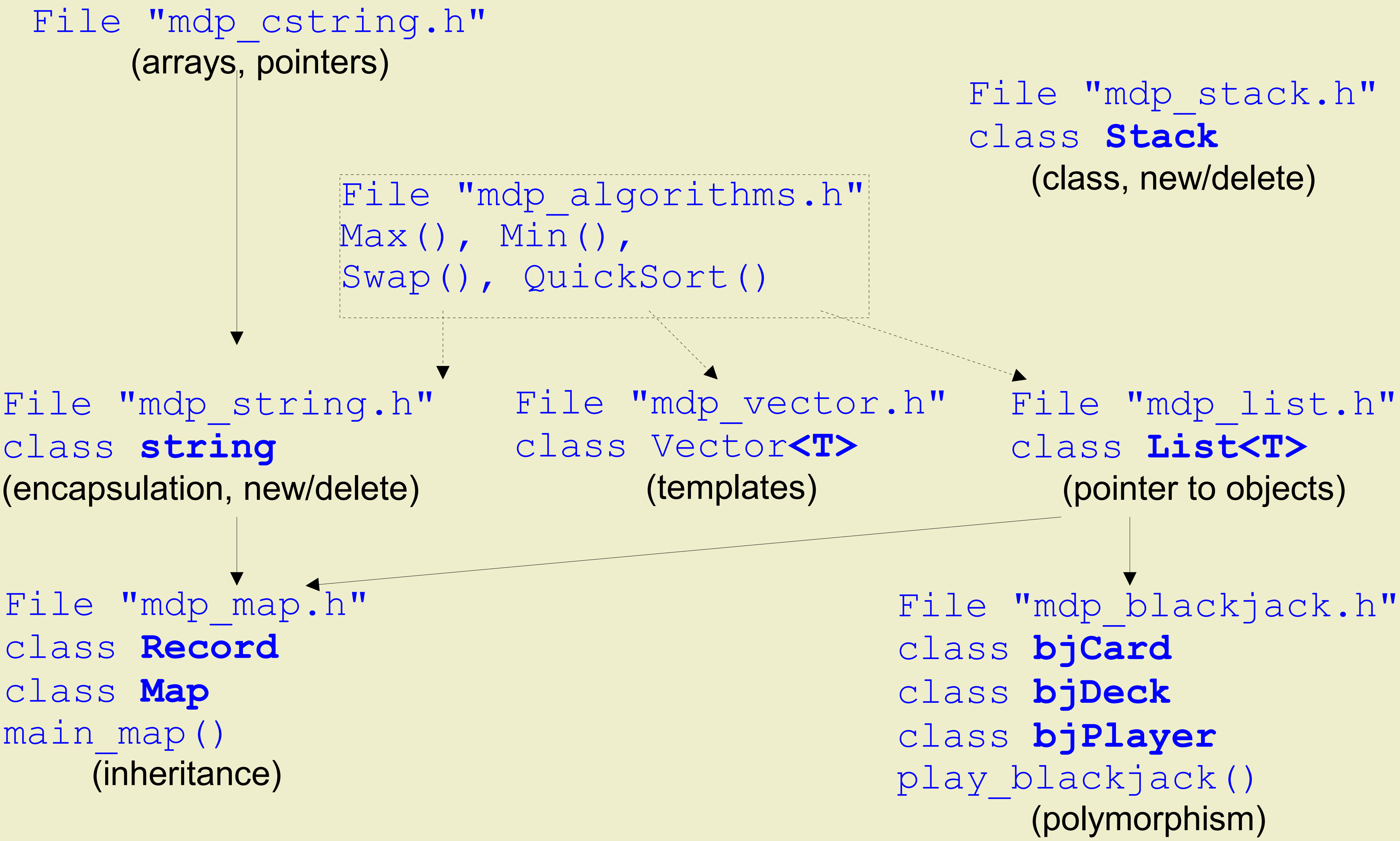
Syllabus

Week 1: Introduction to C++ programming
Week 2: Pointers, arrays and dynamic allocation
Week 3: Encapsulation: array of characters vs class string
Week 4: more on Classes and Objects (class Stack)
Week 5: Classes, Objects and Templates (class Vector, List)
Week 6: **Midterm**
Week 7: Inheritance (class Map)
Week 8: Interfaces and Polymorphism
Week 9: File Input/Output with streams
Week 10: Overview of the Standard Template Libraries

Syllabus (explained)

Week 1: Introduction to C++ programming
(p.1.*,2.1-2.10)
Week 2: Pointers, arrays and dynamic allocation
(p. 3.*,4.1-4.3,4.6,4.9,2.12)
Week 3: Encapsulation: array of characters vs class string
(p. 4.5,4.7,5.1,5.2,5.5,8.3,8.5,9.5, cstring, string)
Week 4: more on Classes and Objects
(p. 5.7,5.9,8.*, class Stack)
Week 5: Classes, Objects and Templates
(p. 8.*,10.1-10.3, class Vector, class List)
Week 7: Inheritance
(p. 6.*, class Map)
Week 8: Interfaces and Polymorphism
(p. 7.*, play Blackjack)
Week 9: File Input/Output with streams
(p. 2.5,2.13, class stream, class fstream)
Week 10: Overview of the Standard Template Libraries

Course hierarchy



File "mdp_cstring.h"
(arrays, pointers)

File "mdp_stack.h"
class **Stack**
(class, new/delete)

File "mdp_algorithms.h"
Max(), Min(),
Swap(), QuickSort()

File "mdp_string.h"
class **string**
(encapsulation, new/delete)

File "mdp_vector.h"
class Vector**<T>**
(templates)

File "mdp_list.h"
class **List<T>**
(pointer to objects)

File "mdp_map.h"
class **Record**
class **Map**
main_map()
(inheritance)

File "mdp_blackjack.h"
class **bjCard**
class **bjDeck**
class **bjPlayer**
play_blackjack()
(polymorphism)

Week 1

Introduction to C++ programming

History of C++

1960: Many computer languages where invented
(including **Algol**)

1970: Ken Thompson invented the **B** language (from Algol)
Norwegian Air Force invented **Simula** and the
concept of **class**

1971: Dennis Ritchie (Bell Labs) invented the **C** as an
extension of the B language.
(90% of Unix was written in C)

1983: Bjarne Stroustrup invented "C with classes". This later became known as **C++**. BS worked at the Computing Laboratory in Cambridge and Bell Labs (now AT&T + Lucent)

Java vs C++

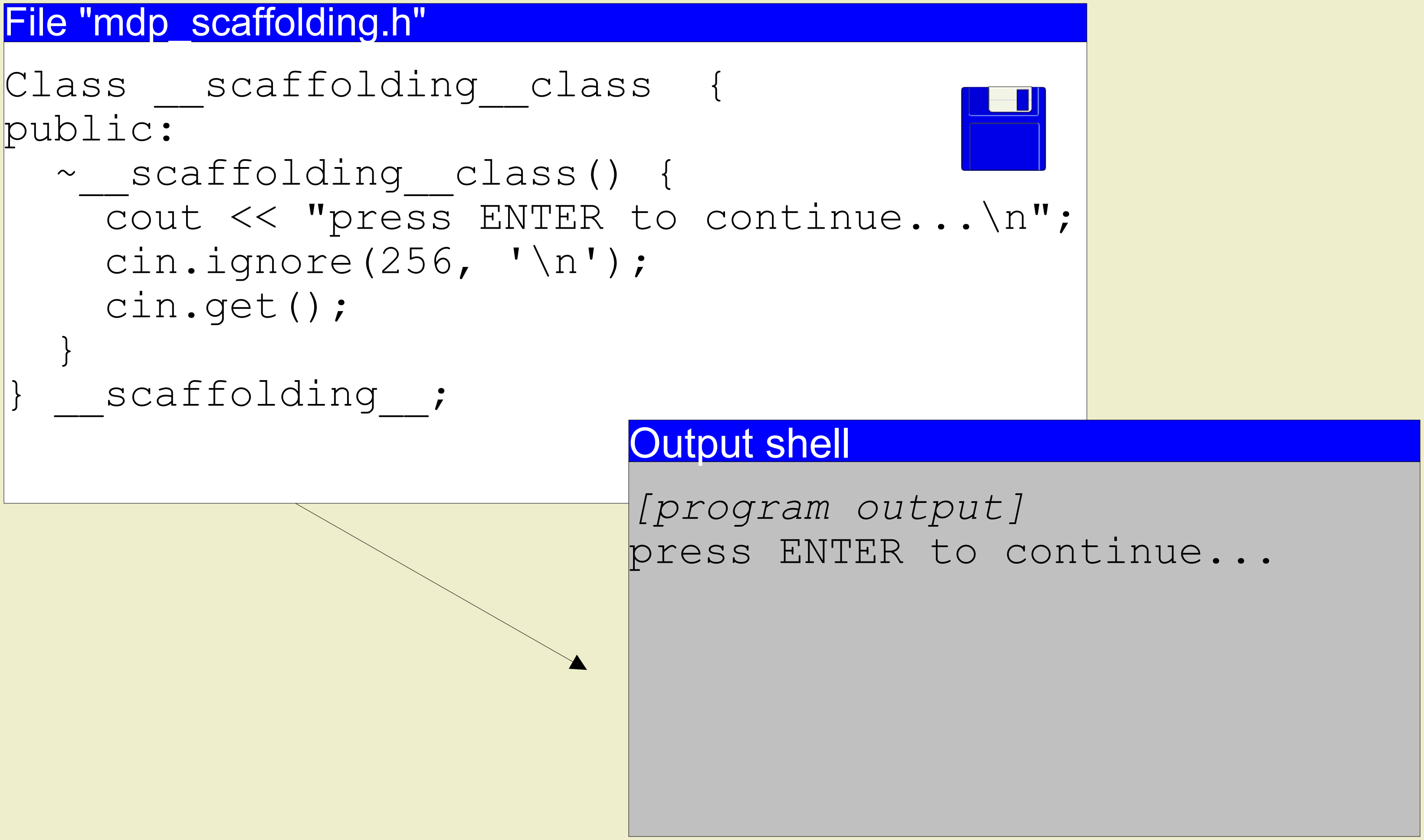
Features	C	C++	Java
portable	y~	y~	y
hardware dependent code	y	y	n
explicit memory management	y	y	n
automatic garbage collection	n~	n~	y
polymorphism	n	y	y
classes	n	y	y
inheritance	n	y	y
multiple inheritance	n	y	n
interfaces	n	y	y
templates	n	y	n
operator overloading	n	y	n
standardized graphic library	n~	n~	y

It is not possible to write an operating system in Java !!!
C++ programs are 2-10 times faster than Java programs !!!

Scaffolding

Tip: Insert scaffolding code at the top of any program after

#include "iostream"



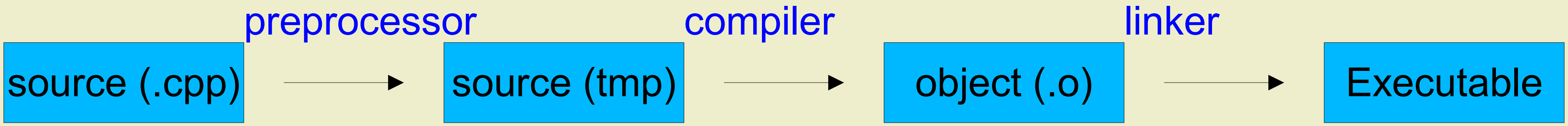
File "mdp_scaffolding.h"

```
Class  __scaffolding__class  {
public:
  ~__scaffolding__class() {
    cout << "press ENTER to continue...\n";
    cin.ignore(256, '\n');
    cin.get();
  }
} __scaffolding__;
```

Output shell

```
[program output]
press ENTER to continue...
```

Typical C++ file structure



preprocessor compiler linker

source (.cpp) → source (tmp) → object (.o) → Executable

File "myprg.cpp"

```
#include "mylib.h"
int main() {
  myfunc(3);
  return 0;
}
```

File "mylib.h"

```
void myfunc(int);
```

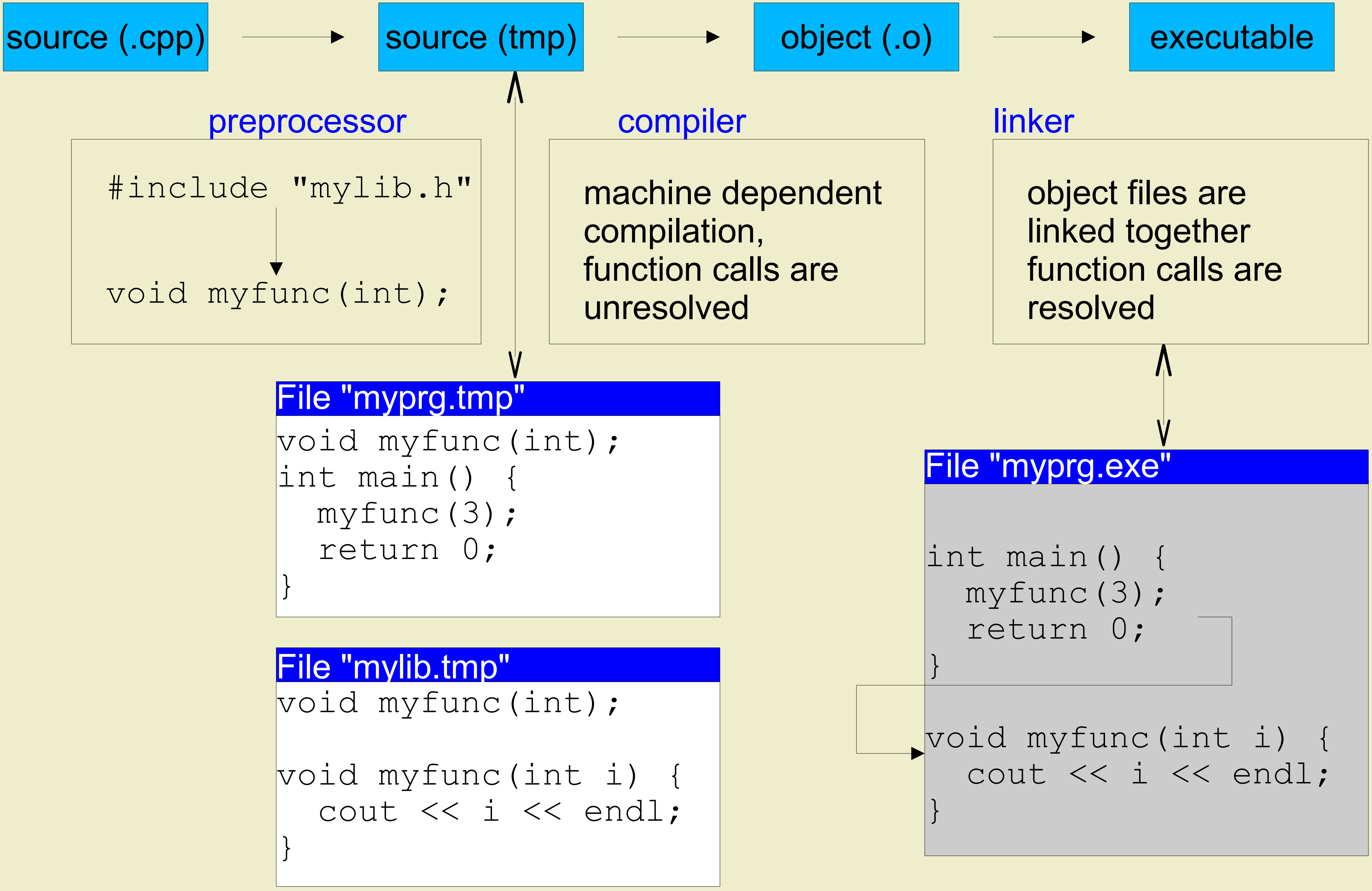
File "mylib.cpp"

```
#include "mylib.h"

void myfunc(int i) {
  cout << i << endl;
}
```

bash shell (cygwin)

```
> ls
mylib.h
mylib.cpp
myprg.cpp

> g++ -ansi -c mylib.cpp -o mylib.o
> ls *.o
mylib.o

> g++ -ansi -c myprg.cpp -o myprg.o
> ls *.o
myprg.o
mylib.o

> g++ myprg.o mylib.o -o myprg.exe
> ./myprg.exe
3
```

Compilation steps



preprocessor

```
#include "mylib.h"

void myfunc(int);
```

compiler

machine dependent compilation, function calls are unresolved

linker

object files are linked together function calls are resolved

File "myprg.tmp"

```
void myfunc(int);
int main() {
   myfunc(3);
   return 0;
}
```

File "mylib.tmp"

```
void myfunc(int);

void myfunc(int i) {
   cout << i << endl;
}
```

File "myprg.exe"

```
int main() {
   myfunc(3);
   return 0;
}

void myfunc(int i) {
   cout << i << endl;
}
```

Function Main

Java Program

```
import java.io.*;

public class HelloWorld {
  public static void main(String args[]) {
    System.out.println("Hello World");
  }
}
```

Program "hello_world_01.cpp"

```
#include "iostream"

int main(int arg, char** args) {
  cout << "Hello World\n";
  return 0;
}
```

In C++ as in Java statements end with semicolon and not with the newline.

Semicolon rules

`#include` is a preprocessor directive rather then a statement.

Preprocessor directives (start with #) are not followed by semicolon

All C++ statements, except preprocessor directives and closed **}** brackets end with semicolon.

Exception: closed **}** bracket terminating a class declaration must be followed by semicolon.

Program "hello_world_01.cpp"

```
#include "iostream"

class myclass { int i; };

int main(int arg, char** args) {
  cout << "Hello World\n";
  return 0;
}
```

Comments

In C and C++ comments can be bounded by */* */* and can be multi line The use of this kind of comment is discouraged since they cannot be nested.

In C++ (not in C) single line comment are indicated with *//*

Program "hello_world_01.cpp"

```
#include "iostream"

/* this is typical
   old style C comment */

// This is a typical
// C++ comment

int main(int arg, char** args) {
  cout << "Hello World\n"; // this is another comment
  return 0;
}
```

Function Main

Program "hello_world_02.cpp"

```
#include "iostream"
int main(int argc, char** argv) {
  cout << "Hello World\n";
  return 0;
}
```

Program "hello_world_03.cpp"

```
#include "iostream"
int main() {
  cout << "Hello World\n";
  return 0;
}
```

Program "hello_world_04.cpp" (deprecated)

```
#include "iostream"
void main() {
  cout << "Hello World\n";
}
```

output shell

```
Hello World
press ENTER to continue...
```

cin and cout (streams and file IO)

Program "cin_01.cpp"

```
#include "iostream"
void main() {
  int i;
  cout << "Type a number\n";
  cin >> i;
  cout << "You typed " << i << endl;
}
```

output shell

```
Type a number
123
You typed 123
press ENTER to continue...
```

Program "file_io_01.cpp"

```
#include "iostream"
#include “fstream”
void main() {
  int i;
  ifstream ifile;
  ifile.open("source.dat");
  ifile >> i;
  ifile.close();
  ofstream ofile;
  ofile.open("destination.dat");
  ofile << "i=" << i << endl;
  ofile.close();
}
```

Types

Program "print_2.cpp"

```
#include "iostream"
void main(int argc, char**argv) {
  int i=2;
  cout << "i=" << i << endl;
}
```

output shell

```
i=2
press ENTER to continue...
```

Program "print_2.3.cpp"

```
#include "iostream"
void main(int argc, char**argv) {
  float i=2.3;
  cout << "i=" << i << endl;
}
```

output shell

```
i=2.3
press ENTER to continue...
```

bool	1bit (?)	true or false
char	8 bits	-128 to 127 or 0 to 255
unsigned char	8 bits	0 to 255
short	16 bits	-32768 to 32767
unsigned short	16 bits	0 to 65535
int	32 bits	same as long
unsigned int	32 bits	unisgned short or unsigned long
long	32 bits	-(~2M) to (~2M)
unsigned long	32 bits	0 to (~4M)
float	32 bits	up to +/- 3.4e+38
double	64 bits	up to +/- 1.8e+308

Assignments are expressions

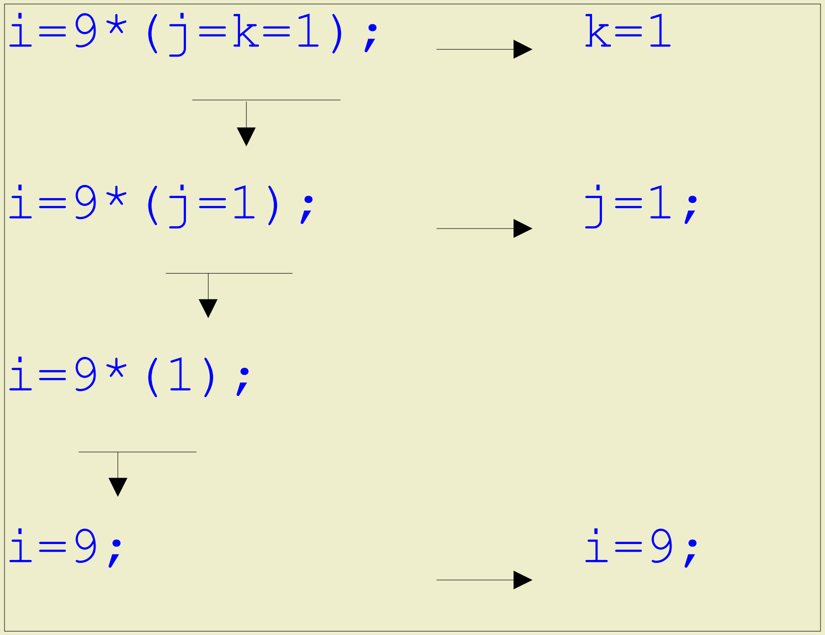
In C++ assignments (i=j) are expressions (i.e. return a value)

Program "assignments_01.cpp"

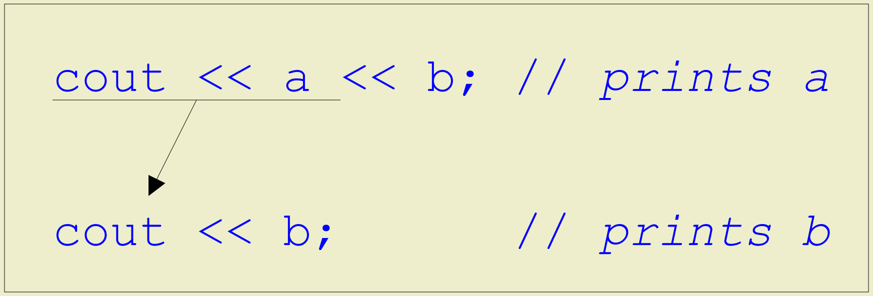
```
#include "iostream"

void main(int argc, char**argv) {
  int i,j,k;

  i=9*(j=k=1);

  cout << i << j << k << endl;
}
```

output shell

```
911
press ENTER to continue...
```



```
i=9*(j=k=1);        k=1

i=9*(j=1);          j=1;

i=9*(1);

i=9;                i=9;
```



```
cout << a << b; // prints a

cout << b;      // prints b
```

for

Program "for_01.cpp"

```
#include "iostream"
void main() {
  int i;
  for(i=0; i<5; i++)
    cout << i << endl;
  cout << "and i=" << i <<endl;
}
```

Program "for_02.cpp"

```
#include "iostream"
void main() {
  int i=0;
  for(; i<5 ;) {
    cout << i << endl;
    i++;
  }
  cout << "and i=" << i <<endl;
}
```

output shell

```
0
1
2
3
4
and i=5
press ENTER to continue...
```

Careful

Program "for_03.cpp"

```
#include "iostream"
void main() {
  for(int i=0; i<5; i++)
    cout << i << endl;
}
```

while

Program "while_01.cpp"

```
#include "iostream"
void main() {
  int i=0;
  while(i<5) {
    cout << i << endl;
    i++;
  }
  cout << "and i=" << i <<endl;
}
```

Program "for_02.cpp"

```
#include "iostream"
void main() {
  int i=0;
  for(; i<5 ;) {
    cout << i << endl;
    i++;
  }
  cout << "and i=" << i <<endl;
}
```

output shell

```
0
1
2
3
4
and i=5
press ENTER to continue...
```

while(*expression*) {...}

equivalent to

for(;*expression*;) {...}

break

Program "while_02.cpp"

```
#include "iostream"
void main() {
  int i=0;
  while(true) {
    cout << i << endl;
    if(++i==5) break;
  }
  cout << "and i=" << i <<endl;
}
```

Program "for_04.cpp"

```
#include "iostream"
void main() {
  int i=0;
  for(i=0; true ;) {
    cout << i << endl;
    if(++i==5) break;
  }
  cout << "and i=" << i <<endl;
}
```

output shell

```
0
1
2
3
4
and i=5
press ENTER to continue...
```

```
i=4;  (i++)==4;
i=4;  (++i)==5;
i=4;  (i--)==4;
i=4;  (--i)==3;
```

if ... else ...

Program "if_01.cpp"

```
#include "iostream"
void main() {
  int i=0;
  if(i==0) cout << "true\n";
}
```

output shell

```
true
press ENTER to continue...
```

Program "if_02.cpp"

```
#include "iostream"
void main() {
  int i=1;
  if(i==0)
    cout << "true\n";
  else
    cout << "false\n";
}
```

output shell

```
false
press ENTER to continue...
```

Program "if_03.cpp"

discouraged

```
#include "iostream"
void main() {
  int i=1;
  cout <<
    ((i==0)?"true\n":"false\n");
}
```

Logical operators
(same as Java)

!	not
&&	and
||	or
==	equal
!=	not equal

Logical values

0	false
1,2,..	true

switch and break

Program "switch_01.cpp"

```
#include "iostream"
void main() {
  int i=0;
  for(i=0; i<5; i++) {
    switch(i) {
      case 0:
        cout << ".\n";
        break;
      case 1:
        cout << "..\n";
        break;
      case 2:
        cout << "...\n";
        break;
      default:
        cout << "default\n";
    }
  }
}
```

output shell

```
.
..
...
default
default
press ENTER to continue...
```

Note: in this example **`break`** breaks the switch statement and not the for loop.

Therefore break is usually required!

switch without break

Program "switch_02.cpp"

```
#include "iostream"
void main() {
  int i=0;
  for(i=0; i<5; i++) {
    switch(i) {
      case 0:
        cout << ".\n";
      case 1:
        cout << "..\n";
      case 2:
        cout << "...\n";
      default:
        cout << "default\n";
    }
  }
}
```

output shell

```
.
..
...
default
..
...
default
...
default
default
default
press ENTER to continue...
```

goto

discouraged

Program "while_switch.cpp"

infinite loop

```
#include "iostream"
void main() {
  int i=0;
  for(i=0;true;i++) {
    switch(i) {
      case 5: break;
    }
    cout << i << endl;
  }
  cout << "end i=" << i <<endl;
}
```

The use of `goto` is discouraged since it is inelegant and never necessary.

To exit nested loops use

`try ... catch ...`

instead...

Program "goto_01.cpp"

discouraged

```
#include "iostream"
void main() {
  int i=0;
  for(i=0;true; i++) {
    switch(i) {
      case 5: goto end_loops;
    }
    cout << i << endl;
  }
end_loops:
  cout << "end i=" << i <<endl;
}
```

output shell

```
0
1
2
3
4
end i=5
press ENTER to continue...
```

try ... catch ... (exceptions)

Program "try_01.cpp"

```
#include "iostream"

void main() {
  int i=0;
  try {
    for(i=0; true; i++) {
      switch(i) {
        case 5: throw 0;
      }
      cout << i << endl;
    }
  } catch(int j) {
    cout << "end i=" << i <<endl;
  }
}
```

output shell

```
0
1
2
3
4
end i=5
press ENTER to continue...
```

```
char* hello="Hello";
try {
  switch(...) {
    case ...: throw 0;
    case ...: throw 1;
    case ...: throw 2;
    case ...: throw hello;
  }
} catch(int j) {
   cout << "j=" << j << endl;
} catch(char* s) {
   cout << "s=" << s << endl;
}
```

try ... catch ... (exceptions)

Program "try_02.cpp"

```
#include "iostream"
#include “mdp_exeption.h”

void main() {
  int i=0;
  try {
    throw Exception(“Whatever”);
  } catch(Exception e) {
    cout << e.value() <<endl;
  }
}
```

output shell

```
Whatever.
```

Function call

Program "global_01.cpp"

```
#include "iostream"

int square(int i) {
  cout << "square called with i=" << i << endl;
  return i*i;
}

void print(int i) {
  cout << "print called with i=" << i << endl;
}

void main() {
  print(square(7));
}
```

- In this example **square** and **print** are global functions (they do not belong to any class and are visible to any other function within the scope (in this case the file)

output shell

```
square called with i=7
print called with i=49
press ENTER to continue...
```

Global variables

Program "global_02.cpp"

discouraged

```
#include "iostream"

int i;

void square() {
  cout << "square called with i=" << i << endl;
  i=i*i;
}

void print() {
  cout << "print called with i=" << i << endl;
}

void main() {
  i=7;
  square();
  print();
  cout << "here i=" << i << endl;
}
```

output shell

```
square called with i=7
print called with i=49
here i=49
press ENTER to continue...
```

In this example **i** is a global variable (it does not belong to any class or function and is visible to any function within the scope (in this case the file)

Passing by value or by reference

Program "by_value.cpp"

wrong

```
#include "iostream"

void swap(int a, int b) {
  int c;
  c=a; a=b; b=c;
}

void main() {
  int i=3, j=4;
  swap(i,j);
  cout << "i=" << i << ", ";
  cout << "j=" << j << endl;
}
```

output shell

```
i=3, j=4
press ENTER to continue...
```

```
memory:   0|0|3|4|0|0|0|3|4|?|0|0
variable:     i j         a b c
```

Program "by_value.cpp"

```
#include "iostream"

void swap(int& a, int& b) {
  int c;
  c=a; a=b; b=c;
}

void main() {
  int i=3, j=4;
  swap(i,j);
  cout << "i=" << i << ", ";
  cout << "j=" << j << endl;
}
```

output shell

```
i=4, j=3
press ENTER to continue...
```

```
memory:   0|0|3|4|0|0|0|0|0|?|0|0
variable:     i j             c
              a b
```

Reference variables

Program "by_value.cpp"

```
#include "iostream"

void main() {
  int i=3;
  int j=i;
  j=4;
  cout << "i=" << i << ", ";
  cout << "j=" << j << endl;
}
```

output shell

```
i=3, j=4
press ENTER to continue...
```

```
memory:   0|0|3|4|0|0|0|0|0|
variable:     i j
```

Program "by_value.cpp"

```
#include "iostream"

void main() {
  int i=3;
  int& j=i;
  j=4;
  cout << "i=" << i << ", ";
  cout << "j=" << j << endl;
}
```

output shell

```
i=4, j=4
press ENTER to continue...
```

```
memory:   0|0|3|0|0|0|0|0|0|
variable:     i
              j
```

Static variables

Program "static_01.cpp"

discouraged

```
#include "iostream"

int j=0;

void increment(int i) {
  cout << "j was " << j;
  j=j+i;
  cout << ", j is " << j << endl;
}

void main() {
  increment(2);
  increment(3);
}
```

output shell

```
j was 0, j is 2
j was 2, j is 5
press ENTER to continue...
```

Program "static_02.cpp"

```
#include "iostream"


void increment(int i) {
  static int j=0;
  cout << "j was " << j;
  j=j+i;
  cout << ", j is " << j << endl;
}

void main() {
  increment(2);
  increment(3);
}
```

output shell

```
j was 0, j is 2
j was 2, j is 5
press ENTER to continue...
```

Returning by reference

Program "static_02.cpp"

```
#include "iostream"

void increment(int i) {
  static int j=0;
  cout << "j was " << j;
  j=j+i;
  cout << ", j is " << j << endl;
}


void main() {
  increment(2);
  increment(3);
}
```

output shell

```
j was 0, j is 2
j was 2, j is 5
press ENTER to continue...
```

Program "static_03.cpp"

advanced

```
#include "iostream"

int& increment(int i) {
  static int j=0;
  cout << "j was " << j;
  j=j+i;
  cout << ", j is " << j << endl;
  return j;
}

void main() {
  increment(2)=10;
  increment(3);
}
```

output shell

```
j was 0, j is 2
j was 10, j is 13
press ENTER to continue...
```

standard libraries

`#include "anything"`

C style	C++ syle	functions
stdio.h	cstdio	printf, scanf, gets, puts, fopen, fclose, fgets, fputf, fwrite, fread, feof, ftell, fseek, ...
string.h	cstring	strlen, strcpy, strcmp, strcat, ...
stdlib.h	ctsdlib	atof, atoi, atol, exit, abort
math.h	cmath	pow, exp, log, sin, cos, ...
complex.h	ccomplex	(*complex numbers*)
time.h	ctime	time, clock
assert.h	cassert	assert
ctype.h	cctype	toupper, tolower
signal.h	csygnal	signal
stdarg.h		(*functions with variable args*)
iostream.h	iostream	(*stream IO functions*)
	string	(*Java like string class*)
	(STL)	(*standard template library*)

Pointers, Arrays and Dynamic Allocation

Memory model, use of &

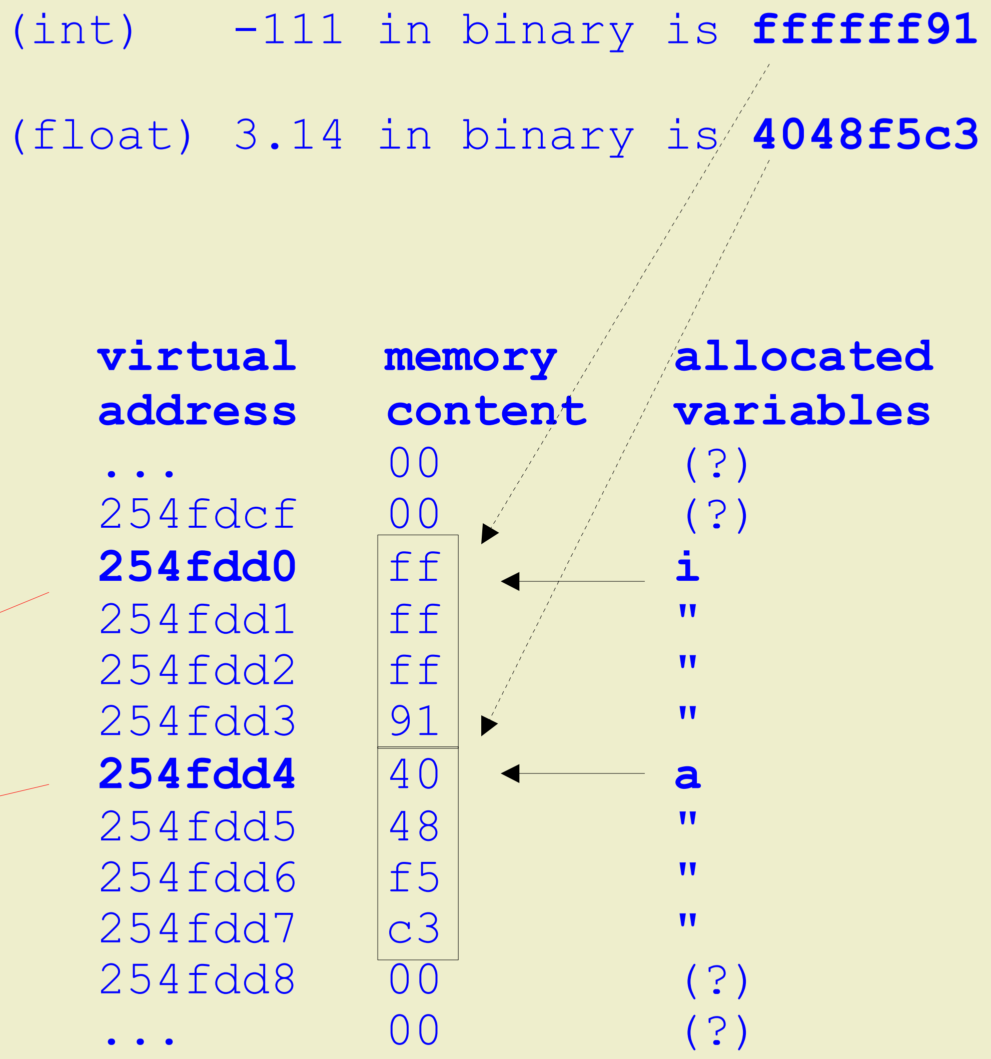
Program "memory_01.cpp"

```
#include "iostream"

void main() {
  int   i=-111;
  float a=3.14;
  cout << i << endl;
  cout << a << endl;
  cout << &i << endl;
  cout << &a << endl;
}
```

&i means **address of i**

output shell

```
-111
3.14
254fdd0
254fdd4
press ENTER to continue...
```



(int)   -111 in binary is **ffffff91**

(float) 3.14 in binary is **4048f5c3**

virtual address	memory content	allocated variables
...	00	(?)
254fdcf	00	(?)
254fdd0	ff	**i**
254fdd1	ff	"
254fdd2	ff	"
254fdd3	91	"
254fdd4	40	**a**
254fdd5	48	"
254fdd6	f5	"
254fdd7	c3	"
254fdd8	00	(?)
...	00	(?)

Memory model
(alternatve notation)

Program "memory_01.cpp"

```
#include "iostream"

void main() {
  int   i=-111;
  float a=3.14;
  cout << i << endl;
  cout << a << endl;
  cout << &i << endl;
  cout << &a << endl;
}
```

`&i` means **address of** `i`

output shell

```
-111
3.14
254fdd0
254fdd4
press ENTER to continue...
```

Ignoring binary representation...

virtual address	memory content	allocated variables
...	?	
254fdcf	?	
254fdd0	-111	**i**
254fdd4	3.14	**a**
254fdd8	?	
...	?	

or equivelent representation

```
address:  ... f 0    4    8 ..
memory:     ?|?|-111|3.14|?|?
variable:       i    a
```

Declaration of pointers

Program "memory_02.cpp"

```
#include "iostream"

void main() {
  int*  p;
  int   i=-111;
  p=&i;

  cout << i << endl;
  cout << sizeof(i) << endl;
  cout << p << endl;
  cout << p+1 << endl;
  cout << p+2 << endl;
}
```

output shell

```
-111
4
254fdd0
254fdd4
254fdd8
press ENTER to continue...
```

int* is type *pointer to integer*

p is declared as a *pointer to integer*

p = *address of* **i**

virtual address	memory content	allocated variables
...	?	
254fda8	254fdd0	**p**
...	...	
254fdd0	-111	**i**
254fdd4	?	
254fdd8	?	
...	?	

Arithmetic of pointers

Program "memory_02.cpp"

```
#include "iostream"

void main() {
  char*  p;
  char   i='c';
  p=&i;

  cout << i << endl;
  cout << sizeof(i) << endl;
  cout << p << endl;
  cout << p+1 << endl;
  cout << p+2 << endl;
}
```

output shell

```
c
1
254fdd0
254fdd1
254fdd2
press ENTER to continue...
```

char* is type *pointer to integer*

p is declared as a *pointer to char*

p = *address of* **i**

virtual address	**memory content**	**allocated variables**
...	?	
254fda8	254fdd0	**p**
...	...	
254fdd0	c	**i**
254fdd1	?	
254fdd2	?	
...	?	

Uses of &

The symbol **&** can be used in four ways:

1) Passing a variable by reference (in the declaration of the arguments of a function)

2) Getting the address of a variable

3) Declaring a variable by refence (i.e. a new name for an existing variable)

4) Returning by refernce

advanced

```
int& func() {
   static int n;
    return n;
}
```

Program "memory_03.cpp"

```
#include "iostream"


void print_address_of(int& k)
    cout << &k << endl;
}

void main() {
  int  i=5;
  int& j=i;
  print_address_of(i);
  print_address_of(j);
}
```

output shell

```
254fdfa
254fdfa
press ENTER to continue...
```

Meaning of *

Program "memory_02.cpp"

```
#include "iostream"

void main() {
   int*  p;
   int   i=5;
   p=&i;
   cout << i << endl;

   *p = 3;

   cout << i << endl;
   cout << p << endl;
}
```

output shell

```
5
3
254fdd0
press ENTER to continue...
```

int* is type *pointer to integer*

p is declared as a *pointer to integer*

p = *address of* **i**

object pointed by **p** = 3

virtual address	**memory content**	**allocated variables**
...	?	
254fda8	254fdd0	**p**
...	...	
254fdd0	2	**i**
254fdd4	?	
254fdd8	?	
...	?	

Uses of *

The symbol * can be used in three ways:

1) Ordinary multiplication

2) Declare a pointer to something

3) Get the object pointed by a pointer

Program "memory_04.cpp"

```
#include "iostream"

void main() {
  float  a=2.4172;
  float* p;

  p=&a;

  *p=3.14159;

  cout << a << endl;
  cout << *p << endl;
}
```

output shell

```
3.14159
3.14159
press ENTER to continue...
```

Casting and conversion

Program "casting_01.cpp"

```
#include "iostream"

void main() {
  float a=3.14159;
  int   i;
  i=(int) a;
  cout << "a=" << a << endl;
  cout << "i=" << i << endl;
}
```

output shell

```
a=3.14159
i=3
press ENTER to continue...
```

float is converted (truncated) to integer

Program "casting_02.cpp"

```
#include "iostream"

void main() {
  float a=3.14159;
  int* p;
  p=(int*) &a;
  cout << "a=" << a << endl;
  cout << "i=" << *p << endl;
}
```

output shell

```
a=3.14159
i=1078530000
press ENTER to continue...
```

The same 32 bits are written as a float and read as an integer

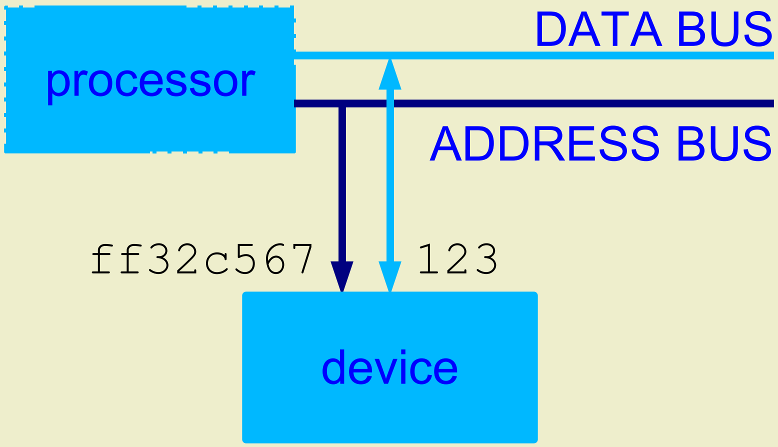
Warning

Programs running in **User mode** should never access memory addresses that were not allocated by the program itself.

This may result in one of the following:

1) a runtime error: ***segmentation fault***
2) corruption of data

Programs running in **Kernel mode** can use pointers to access physical memory and/or devices connected to the system bus.



Program "driver_01.cpp"

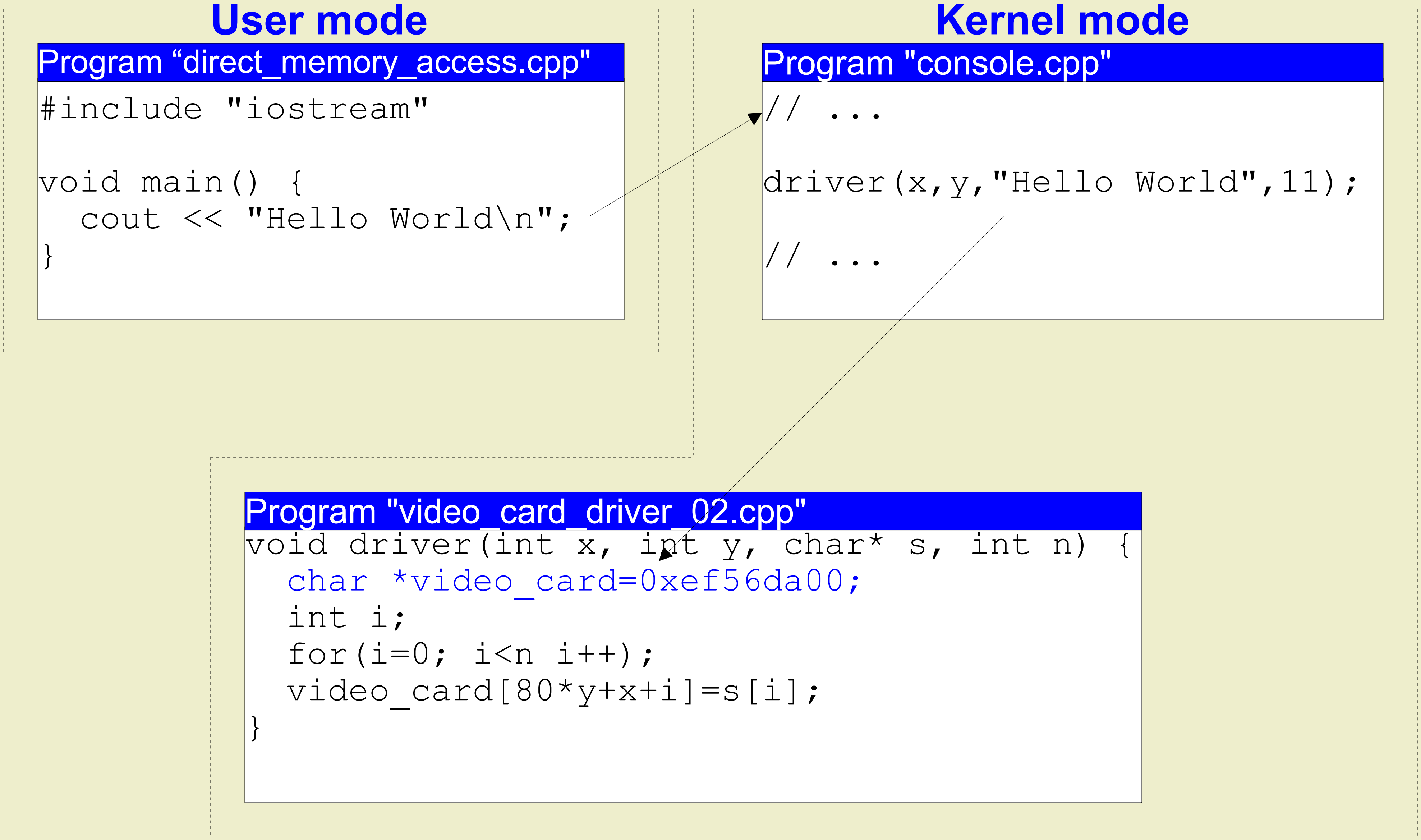
advanced

```
int driver() {
  int* p=0xff32c567
  *p=123;      // write
  return *p;   // read
}
```

0x... indicates that ... is expressed in hexadecimal (a common notation).

advanced

Unix and Windows security



User mode

Program “direct_memory_access.cpp"

```
#include "iostream"

void main() {
  cout << "Hello World\n";
}
```

Kernel mode

Program "console.cpp"

```
// ...

driver(x,y,"Hello World",11);

// ...
```

Program "video_card_driver_02.cpp"

```
void driver(int x, int y, char* s, int n) {
  char *video_card=0xef56da00;
  int i;
  for(i=0; i<n i++);
  video_card[80*y+x+i]=s[i];
}
```

Arrays as Pointers

Program "array_01.cpp"

```
#include "iostream"

void main() {
  int array[3]={2,3,5};
  int *p;
  p=array;
  cout << *p << endl;
  cout << *(p+1) << endl;
  cout << *(p+2) << endl;
}
```

Program "array_02.cpp"

```
#include "iostream"

void main() {
  int array[3]={2,3,5};
  int *p;
  p=array;
  cout << p[0] << endl;
  cout << p[1] << endl;
  cout << p[2] << endl;
}
```

output shell

```
2
3
5
press ENTER to continue...
```

C style arrays are implemented a pointers (even in C++)

Passing arrays

Program "array_03.cpp"

```
#include "iostream"

void set_array(int p[]) {
  p[0]=1; p[1]=2;
  cout << p[0] << p[1] << endl;
}

void main() {
  int a[2]={3,5};
  set_array(a);
  cout << a[0] << a[1] << endl;
}
```

output shell

```
12
12
press ENTER to continue...
```

Program "array_04.cpp"

```
#include "iostream"

void set_array(int* p) {
  p[0]=1; p[1]=2;
  cout << p[0] << p[1] << endl;
}

void main() {
  int a[2]={3,5};
  set_array(a);
  cout << a[0] << a[1] << endl;
}
```

output shell

```
12
12
press ENTER to continue...
```

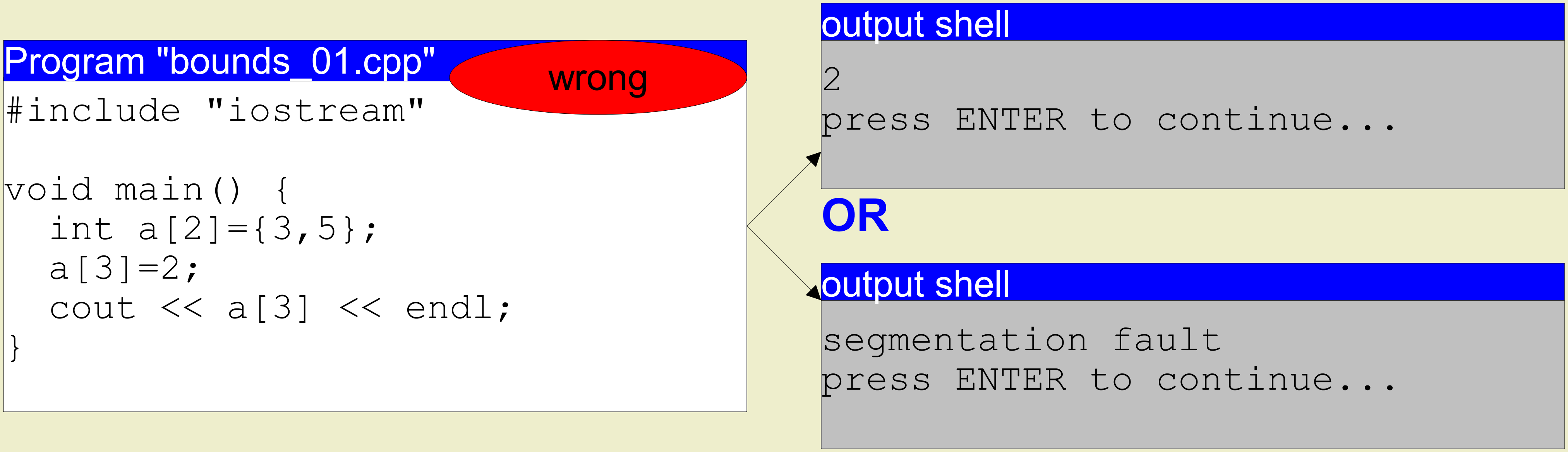
C-style arrays are always passed by reference
(although the pointer to memory can be passed by value or by reference)

Warning

While Java checks for array bounds and eventually return and `ArrayIndexOutOfBoundsException`, **C and C++ do not check for out of bound errors.** In the event this occurs there are two possibilities:

1) The program continues and eventually performs incorrectly.

2) The operative system catches the error and kills the program with a segmentation fault error (the most common error in the history of C/C++).



Program "bounds_01.cpp"

wrong

```
#include "iostream"

void main() {
  int a[2]={3,5};
  a[3]=2;
  cout << a[3] << endl;
}
```

output shell

```
2
press ENTER to continue...
```

OR

output shell

```
segmentation fault
press ENTER to continue...
```

Multidimensional arrays

Program "array_04.cpp"

```
#include "iostream"

const int N=2;

void print_array(int p[N][N]) {
  // access by p[i][j]
}

void main() {
  int a[N][N];
  print_array(a);
}
```

Program "array_05.cpp"

```
#include "iostream"

const int N=2;

void print_array(int* p) {
  // access p[i*N+j]
}

void main() {
  int a[N][N];
  print_array(a);
}
```

Array is always passed by reference
(although the pointer to memory can be passed by value or by reference)

Warning: the notation **int** p** exists but its meaning is different from **int p[][]**. **int** p** means p is pointer to an arrays of pointers to integers. **int p[][]** means a pointer to a 2 dimensional array of integers. **int p[][]** is a pointer of type **int* p**;

More on passing by reference

Program "reference_01.cpp" (only C++)

```
#include "iostream"

void set(int& i) {
  i=3;
}

void main() {
  int j=5;
  set(j);
  cout << j << endl;
}
```

Program "reference_02.cpp" (C style)

```
#include "iostream"

void set(int* p) {
  *p=3;
}

void main() {
  int j=5;
  set(&j);
  cout << j << endl;
}
```

output shell

```
3
press ENTER to continue...
```

The two methods for passing by reference are equivalent.
The pure C++ notation (left window) is cleaner (no use of * and less subject to programmer errors) and, therefore, to be preferred.

Dynamic Allocation

Java Program

```
import java.io.*;

public class HelloWorld {
  public static void main(String args[]) {
    int p[]=new int[3];   // allocation
    for(i=0; i<3; i++) {
      p[i]=i;
      System.out.println(toString(p[i]));
    } // deallocation automatic
  }
}
```

Program "dynamic.cpp"

```
#include "iostream"

void main(int arg, char** args) {
  int* p=new int[3];     // allocation
  for(i=0; i<3, i++) {
     p[i]=i;
     cout << p[i] << endl;
  }
  delete[] p;             // deallocation
}
```

output shell

```
0
1
2
press ENTER to continue...
```

Use of `new` and `delete`

```
class* var = new class[size];
```

Look for **sizeof(*class*)****size*** bytes in memory, ask the Kernel to reserve the memory of the current process and return a pointer to the beginning of that memory. The pointer returned is of type class* and is stored into *var*. *(allocation)*

```
delete[] var;
```

Ask the Kernel to free (for other processes to use) the portion of memory, starting at pointer *var*, that was allocated by this process.
(deallocation)

Remarks:

1) Anything that is allocated must be deallocated.

2) The same memory cannot be deallocated twice. This would result in a runtime error: ***bus error*** (the second most common error in the history of C and C++).

Dynamic Allocation of single objects

Program "dynamic_02.cpp"

```
#include "iostream"

void main(int arg, char** args) {
  int* p=new int;
  *p=5;
  cout << p << endl;
  cout << *p << endl;
  delete p;
}
```

Single Object

`new` *`type`*`;`

↓

`delete` *`addr`*`;`

Program "dynamic_3.cpp"

```
#include "iostream"

void main(int arg, char** args) {
  int* p=new int[2];
  p[0]=5; p[1]=3;
  cout << p << endl;
  cout << p[0] << ", " << p[1] << endl;
  delete[] p;
}
```

Array of Objects

`new` *`type`*`[];`

↓

`delete[]` *`addr`*`;`

Warning using delete

Program "deallocation_01.cp

wrong

```
#include "iostream"

void set(char* p) {
  p[0]='a'; p[1]='b';
  delete[] p;
}

void main() {
  char* s=new char[2];
  set(s);
  cout << s[0] << s[1] << endl;
  delete[] s;
}
```

output shell

```
ab
bus error
press ENTER to continue...
```

Program "deallocation_02.cpp"

```
#include "iostream"

void set(char* p) {
  p[0]='a'; p[1]='b';
}


void main() {
  char* s=new char[2];
  set(s);
  cout << s[0] << s[1] << endl;
  delete[] s;
}
```

output shell

```
ab
press ENTER to continue...
```

C++: new and delete
C   : malloc and delete

Tip: use these new/delete operators for debugging.

Program "mdp_dynalloc.h"

```
#include "malloc.h"
void* operator new(size_t size) {
  cout << "allocating " << size << " bytes";
  void *p=malloc(size);
  cout << " at " << p << endl;
  return p;
}

void operator delete[] (void* pointer) {
  cout << "deallocating from " << pointer << endl;
  free(pointer);
}
```

strict prototypes

OS calls

Program "test_dynalloc_01.cpp"

```
#include "cstdio"
#include "iostream"
#include "dynalloc.h"
void main() {
  float* p=new float[7];
  delete[] p;
}
```

output shell

```
allocating 28 bytes at 0x2670540
deallocating from 0x2670540
press ENTER to continue...
```

Example: average *(passing arrays)*

Program "average.cpp"

```
#include "iostream"
#include "mdp_dynalloc.h"

float average(float* p, long size) {
  float a=0;
  for(int i=0; i<size; i++) a+=p[i];
  return a/size;
}

void main() {
  float *p;
  long size;
  cout << "size="; cin >> size;
  p=new float[size];
  for(int i=0; i<size; i++) {
    cout << "p[" << i << "]=";
    cin >> p[i];
  }
  cout << "average=" << average(p,size) << endl;
  delete[] p;
}
```

output shell

```
size=3
allocating 12 bytes at 0x2670580
p[0]=2
p[1]=3.5
p[2]=1.25
average=2.25
deallocating from 0x2670580
press ENTER to continue...
```

Example: max *(passing arrays)*

Program "max_1.cpp"

```
#include "iostream"
#include "mdp_dynalloc.h"

float max(float* p, long size) {
  float a=p[0];
  for(int i=1; i<size; i++) if(p[i]>a) a=p[i];
  return a;
}

void main() {
  float *p;
  long size;
  cout << "size="; cin >> size;
  p=new float[size];
  for(int i=0; i<size; i++) {
    cout << "p[" << i << "]=";
    cin >> p[i];
  }
  cout << "maximum=" << max(p,size) << endl;
  delete[] p;
}
```

output shell

```
size=3
allocating 12 bytes at 0x2670520
p[0]=2
p[1]=3.5
p[2]=1.25
maximum=3.5
deallocating from 0x2670520
press ENTER to continue...
```

Example: max
(passing and returning arrays)

Program "max_02.cpp"

```
#include "iostream"
#include "mdp_dynalloc.h"

float* input_size(long size) {
  float* p=new  float[size];
  for(int i=0; i<size; i++) {
    cout << "p[" << i << "]=";
    cin >> p[i];
  }
  return p;
}
void max(float* p, long size) {
  float a=p[0];
  for(int i=1; i<size; i++) if(p[i]>a) a=p[i];
  cout << "maximum=" << a << endl;
}
void main() {
  long size;
  cout << "size="; cin >> size;
  float* p=input_float(size);
  max(p, size);
  delete[] p;
}
```

output shell

```
size=3
allocating 12 bytes at 0x2670520
p[0]=10
p[1]=12.45
p[2]=8.16
maximum=12.45
deallocating from 0x2670520
press ENTER to continue...
```

Warning

If there is not enough memory available Java new operator throws an OutOfMemoryException. **C++ new operator throws bad_alloc**

The thrown object should be caught!

Another common practice is to check for the return value of new.

Program "out_of_memory.cpp"

```
#include "iostream"

void main() {
  char* p=new char[100000000000];
  if(p==0)
    cout << "out of memory\n";
  else {
    cout << "memory allocated\n";
    delete[] p;
  }
}
```

output shell

```
memory allocated
press ENTER to continue...
```

OR

output shell

```
out of memory
press ENTER to continue...
```

Exceptions and dynamic allocation

Program "max_03.cpp"

```
#include "iostream"
#include "mdp_dynalloc.h"

float max(float* p, long size)    float a=p[0];
  for(int i=1; i<size; i++) if(p[i]>a) a=p[i];
  return a;
}
void main() {
  float *p;
  long size;
  cout << "size="; cin >> size;
  try {
    p=new float[size];
    if(p==0) throw Exception(“OutOfMe
    for(int i=0; i<size; i++) {
      cout << "p[" << i << "]="; cin >> p[i];
    }
    cout << "maximum=" << max(p,size) << endl;
    delete[] p;
  } catch (Exception e) {
    cout << e.value() << endl;
  }
}
```

output shell

```
size=3
allocating 12 bytes at 0x2670520
p[0]=10
p[1]=12.45
p[2]=8.16
maximum=12.45
deallocating from 0x2670520
press ENTER to continue...
```

output shell

```
size=100000000
allocating 400000000 bytes at 0x0
out of memory
press ENTER to continue...
```

Passing a pointer to pointers

While multidimensional arrays ([][]) can be passed in two ways:

1) by copy

2) by reference (as it were a 1-dimensional array)

pointer to pointers (**) should be passed as such.

While pointers (*) and dynamically allocated arrays (**) have to be deallocated,

regular arrays ([], a in the example) are automatically deallocated.

Program "passing_multiarray.cpp"

```
#include "iostream"

void f(int x[][10]) { };

void g(int* x) { };

void h(int** x) { };

void main() {
  int   a[10][10]
  int** b=new int*[10];
  for(int i=0; i<10; i++)
    b[i]=new int[10];
  f(a);
  g(a);
  h(p);
  // do something ...
  for(int i=0; i<10; i++)
    delete[] b[i];
  delete[] b;
}
```

Week 3

Encapsulation: array of characters vs class string

Array of characters (C-strings)

Java Program

```
import java.io.*;

public class HelloWorld {
  public static void main(String args[]) {
    String s="Hello World";
    System.out.println(s);
  }
}
```

Object

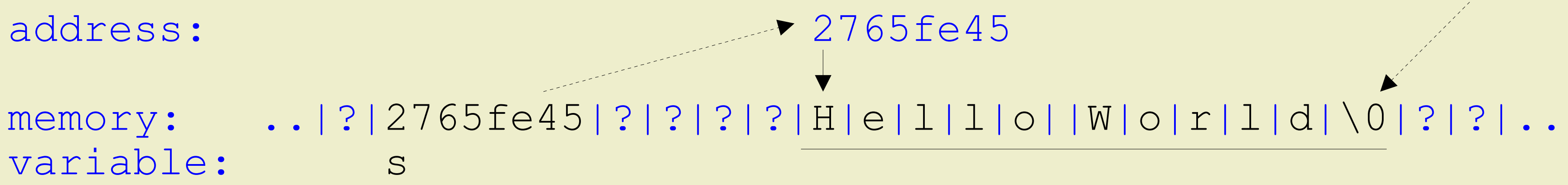
output shell

```
Hello World
press ENTER to continue...
```

Program "array of char 01.cpp"

```
#include "iostream"

void main(int arg, char** args) {
  char* s="Hello World";
  cout << s << endl;
}
```

Pointer to array of characters null ('\0') terminated



```
address:                          2765fe45

memory:    ..|?|2765fe45|?|?|?|?|H|e|l|l|o| |W|o|r|l|d|\0|?|?|..
variable:        s
```

Array of characters (C-strings)

Program "array_of_char_01.cpp"

```
#include "iostream"

void print(char* p) {
  for(;*p!='\0';p++)
    cout << *p;
  cout << endl;
}

void main() {
  char* s="Hello World";
  cout << s << endl;
  print(s);
}
```

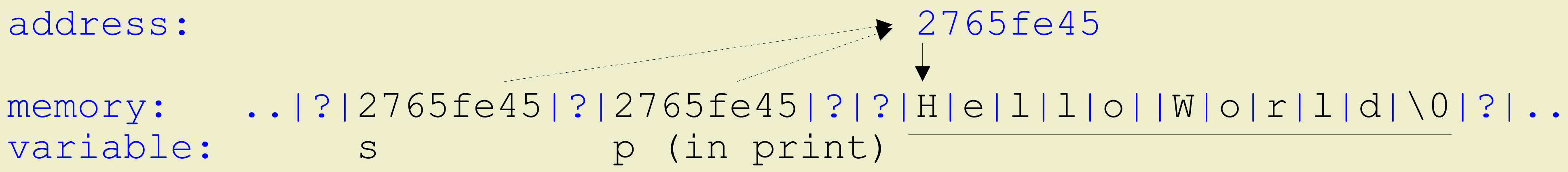
File "mdp_cstring.h"

advanced

```
// ...
ostream& operator<< (ostream& os,
                     char* p) {
  for(;*p!='\0';p++) os << *p;
  return os;
}
// ...
```

output shell

```
Hello World
Hello World
press ENTER to continue...
```



```
address:                                       2765fe45

memory:   ..|?|2765fe45|?|2765fe45|?|?|H|e|l|l|o| |W|o|r|l|d|\0|?|..
variable:      s              p (in print)
```

strlen (length of C-string)

Program "use_strlen_01.cpp"

```
#include "iostream"
#include "cstring"


void main() {
  char* s="Hello World";
  cout << s << endl;
  cout << strlen(s) << endl;
  print(s);
}
```

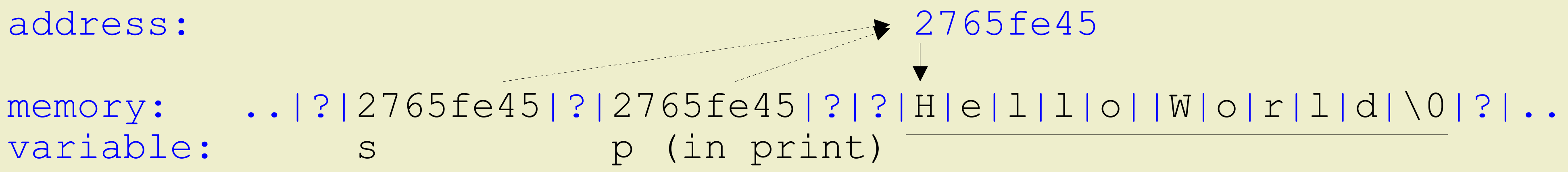
File "mdp_cstring.h"

advanced

```
// ...
int strlen(char* p) {
  int length=0;
  for(;*p!='\0';p++) length++;
  return length;
}
// ...
```

output shell

```
Hello World
11
press ENTER to continue...
```



```
address:                                   2765fe45

memory:   ..|?|2765fe45|?|2765fe45|?|?|H|e|l|l|o| |W|o|r|l|d|\0|?|..
variable:      s           p (in print)
```

strcpy (copy C-strings)

Program "use_strcpy_01.cpp"

```
#include "iostream"
#include "cstring"

void main() {
  char* s="Hello World\n";

  char  r[13];
  strcpy(r,s);
  cout << "r=" << r << endl;

  char* t;
  t=new char[strlen(s)+1];
  strcpy(t,s);
  cout << "t=" << t << endl;
  delete[] t;
}
```

File "mdp_cstring.h"

advanced

```
// ...
void strcpy(char *q, char* p) {
  int i;
  for(i=0; i<strlen(p)+1; i++)
    q[i]=p[i];
}
// ...
```

unsafe

safe

output shell

```
r=Hello World
t=Hello World
press ENTER to continue...
```

strcat (concatenate C-strings)

Program "use_strcat_01.cpp"

```
#include "iostream"
#include "cstring"

void main() {
  char* r="Hello ";
  char* s="World\n";
  int size=strlen(r)+strlen(s)+1;
  char* t=new char[size];
  strcpy(t,r);
  strcat(t,s);
  cout << "r=" << r << endl;
  cout << "s=" << s << endl;
  cout << "t=" << t << endl;
  delete[] t;
}
```

File "mdp_cstring.h"

advanced

```
// ...
void strcat(char *q, char* p) {
  int i, j=strlen(q);
  for(i=0; i<strlen(p)+1; i++)
    q[i+j]=p[i];
}
// ...
```

output shell

```
r=Hello
s=World
t=Hello World
press ENTER to continue...
```

strcmp (compare C-strings)

Program "use_strcmp_01.cpp"

```
#include "iostream"
#include "cstring"


void main() {
  char* r="test\n";
  char* s=new char[strlen(r)+1];
  strcpy(s,r);
  cout << (void*) r << endl;
  cout << (void*) s << endl;
  if(strcmp(r,s)==0)
    cout << "r is equal to s\n";
  else
    cout << "r and s differ\n";
  delete[] s;
}
```

File "mdp_cstring.h"

advanced

```
// ....
int strcmp(char *q, char* p) {
  int i;
  for(i=0; i<strlen(p)+1; i++)
    if(q[i]<p[i])      return -1;
    else if(q[i]>p[i]) return +1;
  return 0;
}
// ...
```

output shell

```
0x401322
0x2670540
r is equal  to s
press ENTER to continue...
```

Passing C-strings

Program "passing_cstrings_01.cpp"

```
#include "iostream"
#include "cstring"

void set(char s[]) {
  strcpy(s, "Hello World");
  cout << s << endl;
}

void main() {
  char s[12]="01234567890";
  set(s);
  cout << s << endl;
}
```

output shell

```
Hello World
Hello World
press ENTER to continue...
```

Program "passing_cstrings_02.cpp"

```
#include "iostream"
#include "cstring"

void set(char* s) {
  strcpy(s, "Hello World");
  cout << s << endl;
}

void main() {
  char s[12]="01234567890";
  set(s);
  cout << s << endl;
}
```

output shell

```
Hello World
Hello World
press ENTER to continue...
```

Command line args as `char**`

Java Program

```
import java.io.*;

public class HelloWorld {
  public static void main(String args[]) {
    int i;
    for(i=0; i<args.length; i++)
      System.out.println(args[i]);
  }
}
```

Program "use_args.cpp"

```
#include "iostream"

int main(int arg, char** args) {
  int i;
  for(i=0; i<argc, i++)
     cout << args[i] << endl;
  return 0;
}
```

output shell

```
>use_args.exe a 3 xx

use_args.exe
a
3
xx
press ENTER to contin
```

Comment on C-stings

C-strings are indispensable in C++ because many libraries use them (for example to pass a filename to a file IO function).

C-strings are unsafe because C++ does not check arrays for out of bounds errors.

Tip: pass a C-string to functions together with the array size and check for out of bound errors.

Program "passing cstrings_03.cpp"

```
#include "iostream"
#include "cstring"

void f(char* s, int size) {
  cout << "array size  = " << size << endl;
  cout << "string size = " << strlen(s) << endl;
}

void main(int arg, char** args) {
  const int N=128;
  char s[N]="aabbccdd";
  f(s, N);
}
```

output shell

```
array size  = 128
string size = 8
press ENTER to continue...
```

Power and Responsibility

"Remember ... with power comes great responsibility"

C++ dynamic allocation is powerful tool but an improper use may easily result in data corruption, incorrect computation and/or runtime errors (segmentation fault, bus error).

Use it responsibly: use encapsulation to hide pointers!

Program "string_01.cpp"

```
#include “mdp_string”

void main(int arg, char** args) {
  String a="Hello";
  String b=" World";
  String c=a+b;
  cout << c << endl;
  cout << "length=" << c.length() << endl;
}
```

output shell

```
Hello World
length=11
press ENTER to continue...
```

First look at classes

Java example of class

```
public class MyClass {
  // member variables

  // constructor

  // other methods
}
```

C++ example of class

```
class MyClass {
public:
  // member variables

  // constructor (allocate stuff)
  // destructor  (deallocate stuff)
  // copy constructor (now to copy class)
  // assignment operator (how assign class)

  // other methods
};
```

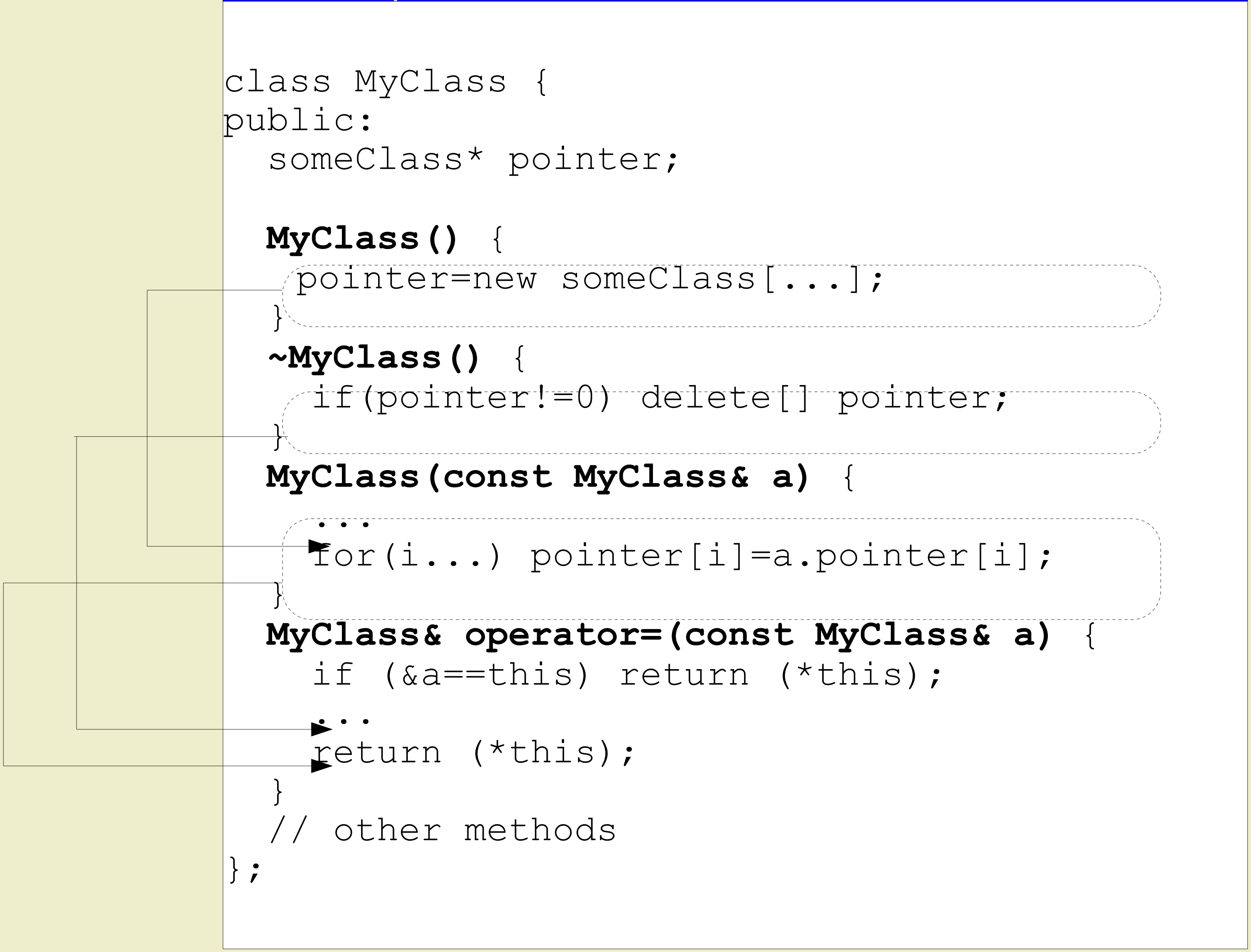
First look at classes

C++ example of class

```
class MyClass {
public:
  // member variables

  MyClass() {
    cout << “constructor: initializing member variables (new)\n”;
  }
  ~MyClass() {
    cout << “destructor: freeing memory (delete)\n”;
  }
  MyClass(const MyClass& a) {
    cout << “copy constructor: copy a into calling object\n”;
  }
  MyClass& operator=(const MyClass& a) {
    cout << “assignment operator: copy a into calling object\n”;
  }
  // other methods
};
```

First look at classes

C++ example of class



```
class MyClass {
public:
  someClass* pointer;

  MyClass() {
    pointer=new someClass[...];
  }
  ~MyClass() {
    if(pointer!=0) delete[] pointer;
  }
  MyClass(const MyClass& a) {
    ...
    for(i...) pointer[i]=a.pointer[i];
  }
  MyClass& operator=(const MyClass& a) {
    if (&a==this) return (*this);
    ...
    return (*this);
  }
  // other methods
};
```

class string

File "mdp_string.h" (constructors/methods overview)

```
class String {
 private:
  char* s;
  int size;
 public:

  String();                           // constructor
  virtual ~String();                  // destructor
  String(const String& p);            // copy constructor
  String& operator=(const String &p); // assignment operator

  String(char* p);                    // converter constructor
  String& operator=(char* p);         // converter assignment

  char* c_str() const;                // other
  int length() const;                 // other
  void resize(int i);                 // other
};
```

For convenience: size = length() + 1 and length() will return size - 1

constructor/destructor

File "mdp_string.h"

```
String() {
  cout << "call to constructor\n";
  s=new char[size=1];
  s[0]='\0';
}

~String() {
  cout << "call to destructor\n";
  delete[] s;
}

void resize(int i) {
  cout << "  call to resize\n";
  if(s!=0) delete[] s;
  s=new char[size=i+1];
  s[0]='\0';
}
```

constructor

destructor

Program "test_string_01.cpp"

```
void main() {
  String a;
  a.resize(100);
}
```

output shell

```
call to constructor
  call to resize
call to destructor
press ENTER to
continue...
```

copy constructors

File "mdp_string.h"

```
Int length() const {
  return size-1;
}

String (const string& p) {
  cout << "call to c.c.\n";
  size=0; s=0;
  resize(p.length());
  strcpy(s,p.s);
}

String (char* p) {
  cout << "call to converter\n";
  size=0; s=0;
  resize(strlen(p));
  strcpy(s,p);
}
```

string length

copy constructor

converter

Program "test_string_02.cpp"

```
void main() {
  String a("This is a test");
}
```

output shell

```
call to converter
  call to resize
call to destructor
press ENTER to
continue...
```

assignment operator

File "mdp_string.h"

```
String& operator= (const String& p) {
  cout << "operator=(string)\n";
  if(&p==this) return (*this);
  resize(p.length());
  strcpy(s,p.s);
  return *this;
}
String& operator= (char* p) {
  cout << "operator=(char*)\n";
  resize(strlen(p));
  strcpy(s,p);
  return *this;
}
char* c_str() const {
  return s;
}
```

assignment operators

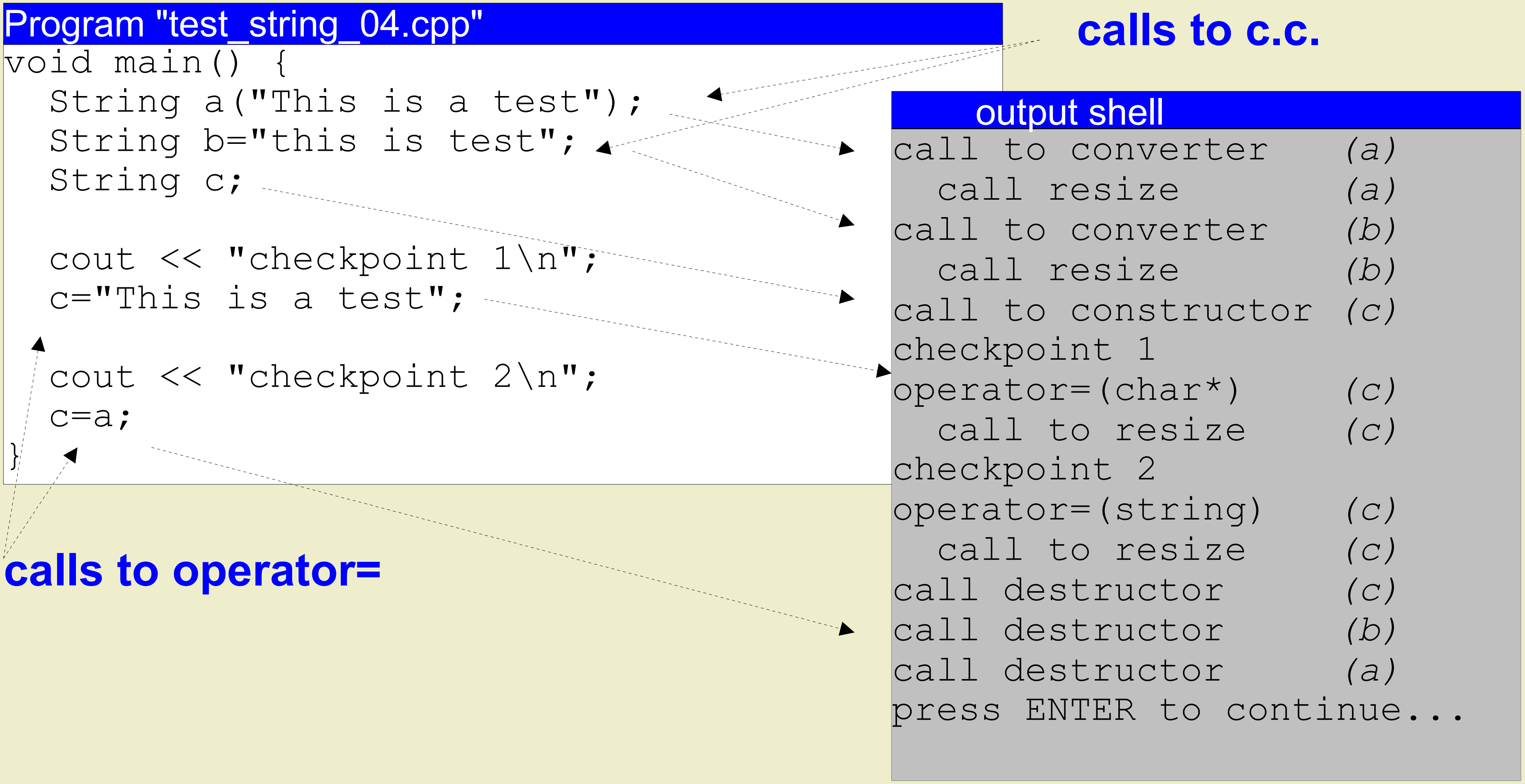
Program "test_string_03.cpp"

```
void main() {
  String a;
  a="This is a test";
  cout << a.c_str() << endl;
}
```

output shell

```
call to constructor
operator=(char*)
  call to resize
This is a test
call to destructor
press ENTER to
continue...
```

More on copy constructor



Program "test_string_04.cpp"

```
void main() {
  String a("This is a test");
  String b="this is test";
  String c;

  cout << "checkpoint 1\n";
  c="This is a test";

  cout << "checkpoint 2\n";
  c=a;
}
```

calls to c.c.

calls to operator=

output shell

```
call to converter   (a)
  call resize       (a)
call to converter   (b)
  call resize       (b)
call to constructor (c)
checkpoint 1
operator=(char*)    (c)
  call to resize    (c)
checkpoint 2
operator=(string)   (c)
  call to resize    (c)
call destructor     (c)
call destructor     (b)
call destructor     (a)
press ENTER to continue...
```

Note that the symbol = following a class declaration constitutes a call to the copy constructor (c.c. or the converter) and not `operator=`

example of iostream

File "mdp_string.h"

```
ostream& operator<<(ostream &os, String p) {
  os << p.c_str();
  return os;
}

istream& operator>>(istream& is, const String& p){
  static char buffer[1024];
  if(is.peek()=='\n') is.ignore(1,'\n');
  is.getline(buffer, 1024);
  p=buffer; //here buffer is copied in p
  return is;
}
```

input/output for arbitrary stream

Attention: different from standard string!

Program "test_string_05.cpp"

```
void main() {
  String a;
  cin >> a;
  cout << "you typed\n";
  cout << a << endl;
}
```

output shell

```
call to constructor    (a)
nothing to say
operator=(char*)
  call to resize       (p)
you typed
nothing to say
call to destructor
press ENTER to continue...
```

more on operator overloading

File "mdp_string.h"

```
string operator+(const String& a,
                 const String& b) {
  String c;
  c.resize(a.length()+b.length());
  strcpy(c.c_str(),a.c_str());
  strcat(c.c_str(),b.c_str());
  return c;
}
```

Program "test_string_06.cpp"

```
void main() {
  String a,b,c;
  cin >> a;
  cin >> b;
  c=a+b;
  cout << c;
}
```

output shell

```
[comments removed]
aaaa
bbbb
aaaabbbb
press ENTER to continue...
```

more on operator overloading

File "mdp_string.h"

```
bool operator==(const String& a,
                const String& b) {
  if(strcmp(a.c_str(),b.c_str())==0)
    return true;
  return false;
}

bool operator!=(const String& a,
                const String& b) {
  if(strcmp(a.c_str(),b.c_str())==0)
    return false;
  return true;
}
```

Program "test_string_06.cpp"

```
void main() {
  String a,b;
  cin >> a;
  cin >> b;
  if(a==b) cout << "==\n";
  if(a!=b) cout << "!=\n";
}
```

output shell

```
[comments removed]
aaaa
bbbb
!=
press ENTER to continue...
```

more on operator overloading

File "mdp_string.h"

```
bool operator<(const String& a,
               const String& b) {
  return (strcmp(a.c_str(),b.c_str())<0);
}

bool operator>(const String& a,
               const String& b) {
  return (strcmp(a.c_str(),b.c_str())>0);
}

bool operator<=(const String& a,
                const String& b) {
  return (strcmp(a.c_str(),b.c_str())<=0);
}

bool operator>=(const String& a,
                const String& b) {
  return (strcmp(a.c_str(),b.c_str())>=0);
}
```

Program

```
void main() {
  String a,b;
  cin >> a;
  cin >> b;
  if(a<b)
    cout << "<\n";
  if(a>b)
    cout << ">\n";
}
```

output shell

```
aaaa
aaba
<
press ENTER to continue...
```

The built-in class **string**

Functions `strlen, strcpy, strcmp` and `strcat` are standard C/C++ libraries and are declared in the header *"cstring"* or *"string.h"*

`class string` is now standard in C++ and is declared in the header *"string"*. In this lectures we built a similar class string in the file *"mdp_sstring.h"*

The class string that we use in these lectures is based on 8bits characters and not 16bits wide characters (**`wchar_t`**) as in Java. Although the latter exists in C++ its use is not common. This difference must be take into account when transferring strings from Java to C/C++ and vice versa.

Program "test_string_07.cpp"

```
#include "iostream"
#include "mdp_string"
using namespace std;
void main() {
  String a,b,c;
  cin >> a;
  cin >> b;
  c=a+b;
  cout << c << endl;
}
```

in Visual C++
this is required to use
class string (ANSI '97)

Passing a class by reference

Through encapsulation and operator overloading C++ allows us to extend the language.

As for int or float, a string object can be passed by reference or by value (with a call to c.c.).

Program "test_string_08.cpp"

```
#include "iostream"
#include "mdp_string.h"

void f(String s) {
  s="bbb";
}
void main() {
  String a="aaa";
  f(a);
  cout << a << endl;
}
```

output shell

```
aaa
press ENTER to continue...
```

Program "test_string_09.cpp"

```
#include "iostream"
#include "mdp_string.h"

void g(String& s) {
  s="bbb";
}
void main() {
  String a="aaa";
  g(a);
  cout << a << endl;
}
```

output shell

```
bbb
press ENTER to continue...
```

Returning a class by reference

A class string (as any other class) should not be returned by reference unless 1) one returns ***this**; 2) one returns an argument of that was passed by reference; 3) one returns a static local variable.

Program "test_string_10.cpp"

wrong

```
#include "iostream"
#include "mdp_string.h"

string& f() {
  return String("Hello");
}
void main() {
  cout << f() << endl;
}
```

output shell

```
>g++ test_string_10.cpp
Error:
initialization of non-const
```

Program "test_string_11.cpp"

```
#include "iostream"
#include "mdp_string.h"

string g() {
  return String("Hello");
}
void main() {
  cout << g() << endl;
}
```

output shell

```
Hello
press ENTER to continue...
```

Typical classes

The class T declared below is typical for almost any class one may need,

If the class member variables do not include pointers it is possible to omit the destructor, the copy constructor and the assignment operator from the declaration, since the default ones are probably good enough (class S).

Program "class T.cpp"

```
class T {
 private:
  // member variables and member pointers
 public:
  T();                    // constructor
  virtual ~T();           // destructor
  T(const T&);            // copy constructor
  T operator=(const T&);  // assignment op.
  // member functions
};

ostream& operator<<(ostream& os,const T&);
bool operator==(const T&, const T&);
bool operator!=(const T&, const T&);
// other operators ... and functions
```

Program "class S.cpp"

```
class S {
 private:
  // member variables
  // no pointers
 public:
  S();
 // member functions
};




ostream& operator <<
(ostream& os,const S&);
// other operators
// and functions
```

My rules

Most important rules of C++ programming:

Everything that is allocated must be deallocated (only once)

One should not pass or return by value an object that contains pointer(s), unless one has properly explicitly defined the copy constructor.

One should not call the assignment operator (=) of an object that contains pointer(s) unless one has explicitly properly defined the assignment operator.

Failure to comply with these rules will result in memory leaks, runtime errors or wrong results!

Week 4

Classes and Objects (class Stack)

Hello world with objects

Java Program

```
import java.io.*;

public class HelloWorld {
  public static void main(String args[]) {
    System.out.println("Hello World");
  }
}
```

Program "hello_world_02.cpp"

```
#include "iostream"

class HelloWorld {
public:
  static void main(int argc, char** args) {
    cout << "Hello World\n";
  }
};

void main(int argc, char** args) {
  HelloWorld::main(argc, args)
}
```

class and struct

In C++ class and struct are similar except that class members are private by default (unless otherwise specified) while struct members are public by default (unless otherwise specified).

Program "class vs struct 01.cpp"

```
#include "iostream"

class A {
  void m() {}
};

struct B {
  void m() {}
};
```

by default private:

by default public:

CSC 309 – OOP in C++
Prof. Massimo Di Pierro

public, protected, private

Program "access_modifiers_01.cpp"

```
#include "iostream"
class A {
public:
  int one() { return 1; }
protected:
  int two() { return 2; }
private:
  int three() { return 3; }
};

class B : public A {
public:
  int four() { return two()+one()+1; }
};

void main() {
  A a;
  B b;
  cout << a.one() << endl;
  cout << b.one() << endl;
  cout << b.four() << endl;;
}
```

visible everywhere

only visible within the class and within derived classes

only visible within the class cannot be inherited

members of class B see one() and two() not three()

Only A::one(), B::one() and B::four() are visible.

static and inline

Program "static and inline 01.cpp"

```
#include "iostream"

class A {
public:
  static int one() { return 1; }
  int two() { return 2; }
  inline int three() { return 3; }
};

void main() {
  cout << A::one() << endl;

  A a;
  cout << a.two() << endl;
  cout << a.three() << endl;
}
```

static member functions can be called even if the class is not instantiated

inlined functions are expanded inline by the compiler without function call. To be used for speed.

friend

Program "friend 01.cpp"

wrong

```
#include "iostream"

class IntC {
private: int i;
public:
   void set(int j) { i=j };
   int  get() { return i; };
};

void print(IntC& a) {
   cout << a.i << endl;
}
void main() {
   IntC a;
   a.set(123);
   print(a);
}
```

Program "friend 02.cpp"

```
#include "iostream"

class IntC {
private: int i;
public:
   void set(int j) { i=j };
   int  get() { return i; };
   friend void print(IntC& a) {
      cout << a.i << endl;
   }
};

void main() {
   IntC a;
   a.set(123);
   print(a);
}
```

friend functions are called as ordinary functions (not methods) but can access private member variables and methods.

Example: class Stack

Program "test_stack_01.cpp"

```
#include "iostream"
#include "mdp_stack.h"

void main() {
  Stack a;
  for(int i=0; i<5; i++) a.push(i);
  for(int i=0; i<5; i++) cout << a.pop(i);
}
```

output shell

```
43210
press ENTER to continue...
```

File "mdp_stack.h" (constructors / methods overview)

```
class Stack {
 public:
  enum {MaxStack = 5 }; // constant!
  Stack();
  bool isEmpty() const;
  bool isFull() const;
  void push(int n);
  int  pop();
  friend ostream& operator<<(ostream& os, const Stack& s);
 private:
  int top;  // pointer within the stack
  int arr[MaxStack]; // stack container
};
```

Example: class Stack

File "mdp_stack.h" (continue)

```
Stack() { top=-1; }
bool isEmpty() const { return top < 0; }
bool isFull() const { return top == MaxStack-1; }
void push(int n) {
  if(isFull())
    throw Exception("StackFullException");
  else
    arr[++top]=n;
}
int pop() {
  if(isEmpty())
    throw Exception("StackEmptyException");
  return arr[top--];
}
ostream& operator<<(ostream& os, const Stack& s) {
  if(s.isEmpty()) os << "[]";
  else {
    os << "[";
    for(int i=s.top; i>=0; i--) os << i << ":" << s.arr[i] << " ";
    os << "]";
  }
  return os;
}
```

Example: better class Stack

```
void main() {
  Stack a(10);
  for(int i=0; i<10; i++) a.push(i);
  for(int i=0; i<10; i++) cout << a.pop(i);
}
```

output shell

```
9876543210
press ENTER to continue...
```

File "mdp_stack.h"

```
class Stack {
public:
  Stack(int i=5);
  ~Stack();
  Stack(const Stack& s);
  Stack& operator=(const Stack& s);
  bool isEmpty() const;
  bool isFull() const;
  void push(int n);
  int  pop();
  friend ostream& operator<<(ostream& os, const Stack& s);
private:
  int  MaxStack;
  int  top; // pointer within the stack
  int* arr; // stack container
};
```

File "mdp_stack.h" (continue)

```
Stack(int i) {
  arr=new int[MaxStack=i];
  top=-1;
}

~Stack() {
  delete[] arr;
}
```

Example: better class Stack

File "mdp_stack.h" (continue)

```
Stack(const Stack& s) {
  top=s.top;
  MaxStack=s.MaxStack;
  arr=new int[MaxStack];
  for(int i=0; i<MaxStack; i++)
    arr[i]=s.arr[i];
}
operator=(const Stack& s) {
  if(&s==this) return (*this);
  delete[] arr;
  top=s.top; MaxStack=s.MaxStack;
  arr=new int[MaxStack];
  for(int i=0; i<MaxStack; i++)
    arr[i]=s.arr[i];
  return *this;
}
```

Copy Constructor

Assignment operator

```
void main() {
  Stack s1(10); // constructor call
  for(int i=0; i<10; i++) s1.push(i);
  Stack s2=s1;  // c.c call
  for(int i=0; i<10; i++) cout << s2.pop();
}
```

output shell

```
9876543210
press ENTER to continue...
```

this and *this

The keyword **`this`**, used within a class, is a pointer to the present object (instantiation of the class).

`*this` is the object itself.

The assignment operator must **`return (*this);`**

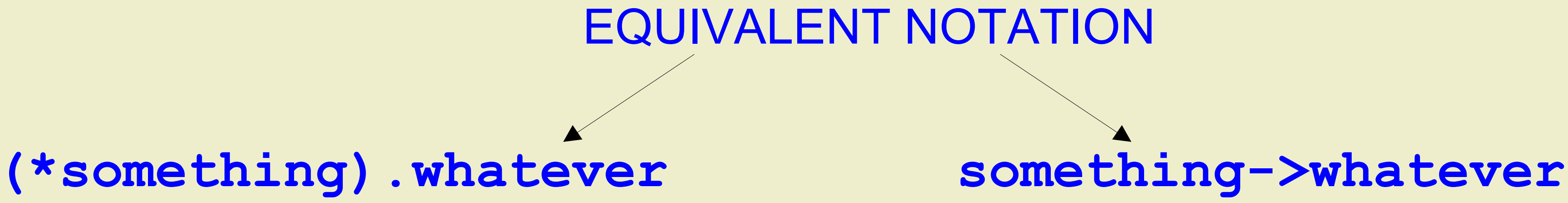
```
class T {
  T& operator=(...) {
    ...
   return (*this);
}
```

```
(*this).operator[](i)=a;
```

The keyword **`this`** is also commonly used to call overloaded operators

```
class T {
  float a[100];
  float& operator[](int i) {
    return a[i];
  }
  void set(int i, float a) {
    (*this)[i]=a;
  }
}
```

operator ->



EQUIVALENT NOTATION

(*something).whatever

something->whatever

```
#include "iostream"

class Euclid {
 public:
  float pi() {
    return 3.14159;
  }
}

void main() {
  Euclid* x=new Euclid;
  cout << (*x).pi() << endl;
  delete x;
}
```

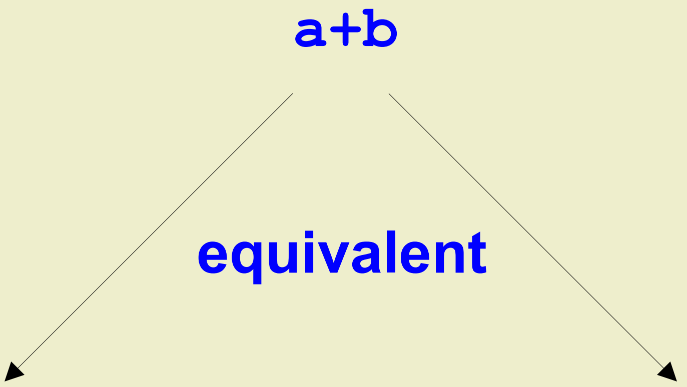
```
#include "iostream"

class Euclid {
 public:
  float pi() {
    return 3.14159;
  }
}

void main() {
  Euclid* x=new Euclid;
  cout << x->pi() << endl;
  delete x;
}
```

Week 5

Classes, Objects and Templates (class Vector, List)

Overloading binary operators

```
class A {
  int value[100];
} a,c;
class B {
  int value[100];
} b;
```



operator+(a,b)

a.operator+(b)

```
A operator+(const A& a,
            const B& b) {
  A c;
  for(int i=0; i<100; i++)
    c.value[i]=
      a.value[i]+b.value[i];
  return c;
}
```

```
A A::operator+(const B& b) {
  A c;
  for(int i=0; i<100; i++)
    c.value[i]=
      value[i]+b.value[i];
  return c;
}
```

Same for any **operator@** where **@** can be any of the following:

new new[] delete delete[] + - * / % | &
>> << >>= <<= > < >= <= == != && || -> ->* ,

Name Overloading

Java Program

```
import java.io.*;

public class TestOverloading {
  public static int square(int x) {
    return x*x;
  }
  public static float square(float x) {
    return x*x;
  }
  public static main() {
    int i=2;
    float a=3.1415926535897;
    System.out.println(toString(square(i)));
    System.out.println(toString(square(a)));

  }
}
```

Name Overloading

Program "overloading_01.cpp"

```
#include "iostream"

int square(int x) {
  return x*x;
}

float square(float x) {
  return x*x;
}

void main() {
  int i=2;
  float a=3.1415926535897;
  cout << square(i)<< endl;
  cout << square(a)<< endl;
}
```

different functions:
same names but different
arguments and different bodies

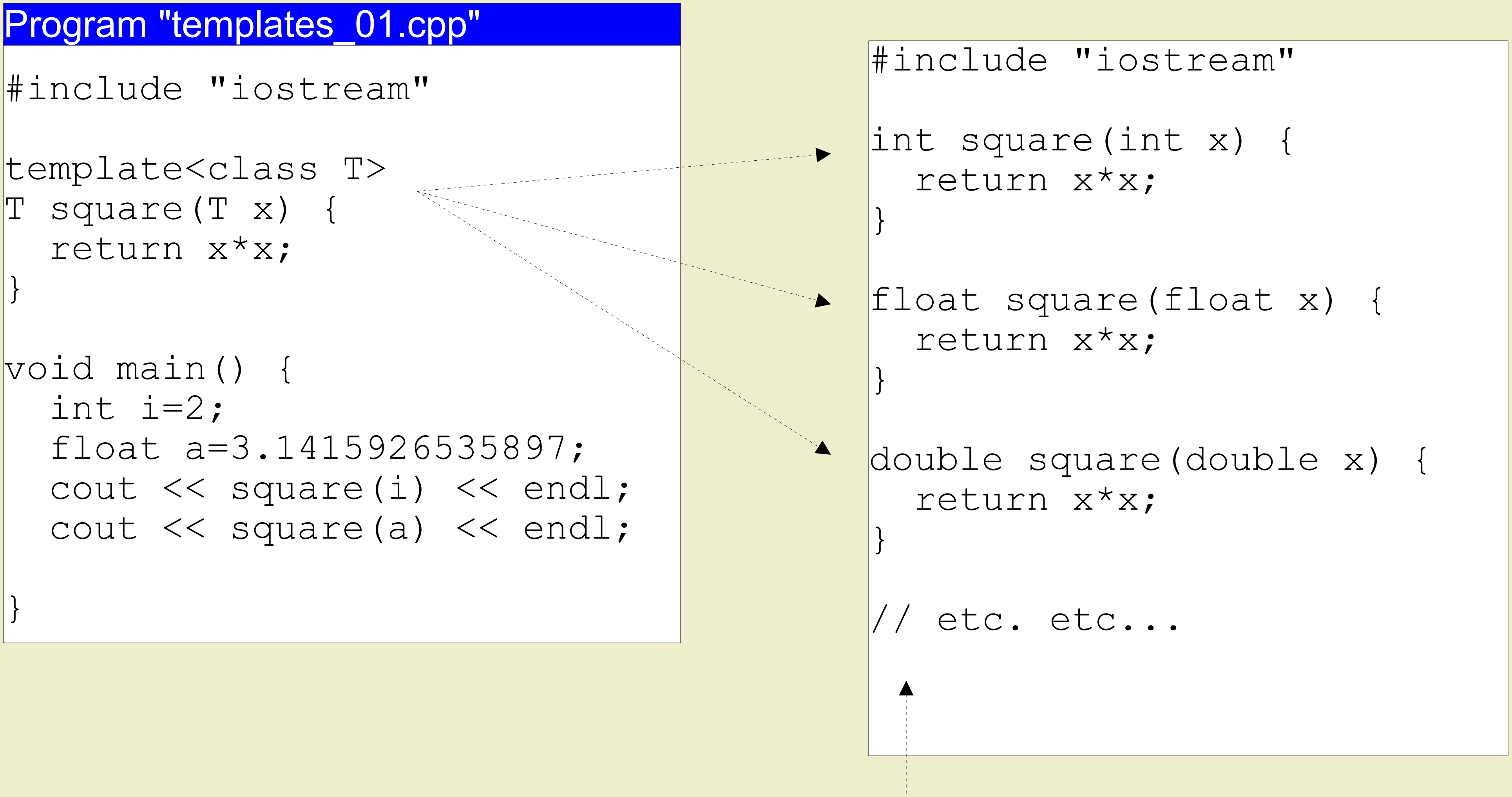
Name Overloading and weak polymorphism`

Program "overloading_02.cpp"

```
#include "iostream"

class C {
 public:
  static int square(int x) {
    return x*x;
  }
  static float square(float x) {
    return x*x;
  }
};

void main() {
  int i=2;
  float a=3.1415926535897;
  cout << C::square(i) << endl;
  cout << C::square(a) << endl;
}
```

different functions:
same names but different
arguments and different bodies

Templates



Program "templates_01.cpp"

```
#include "iostream"

template<class T>
T square(T x) {
  return x*x;
}

void main() {
  int i=2;
  float a=3.1415926535897;
  cout << square(i) << endl;
  cout << square(a) << endl;

}
```

```
#include "iostream"

int square(int x) {
  return x*x;
}

float square(float x) {
  return x*x;
}

double square(double x) {
  return x*x;
}

// etc. etc...
```

different functions:
same names and same bodies but
different argument types

Example: min and max

File "mdp_algorithms.h"

```
template<class T>
T Min(const T& a, const T& b) {
  if(a<b) return a;
  return b;
}
```

File "mdp_algorithms.h"

```
Template<class T>
T Max(const T& a, const T& b) {
  if(a>b) return a;
  return b;
}
```

Program "test_min_max_01.cpp"

```
#include "iostream"
using namespace std;
#include "mdp_algorithms.h"

void main() {
  int a=3;
  int b=5;
  cout << Min(a,b) << endl;
  cout << Max(a,b) << endl;
}
```

output shell

```
3
5
press ENTER to continue...
```

Example: swap with templates

File "mdp_algorithms.h"

```
template<class T>
void Swap(T& a, T& b) {
  T c=a;
  a=b;
  b=c;
}
```

Program "test_swap_01.cpp"

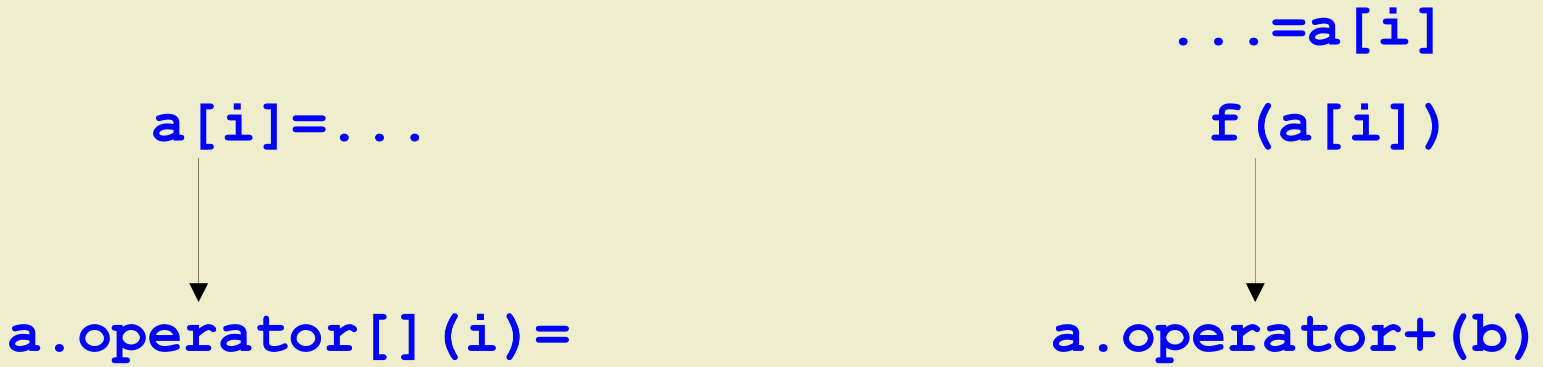
```
#include "iostream"
using namespace std;
#include “mdp_string.h”
#include "mdp_algorithms.h"

void main() {
  String a="Hello";
  String b="World";
  Swap(a,b);
  cout << a << endl;
  cout << b << endl;
}
```

output shell

```
World
Hello
press ENTER to continue...
```

Overloading operator[] and/or operator()

```
class A {
  int value[100];
} a;
```



`...=a[i]`

`a[i]=...`

`f(a[i])`

`a.operator[](i)=`

`a.operator+(b)`

```
int& A::operator[](int i)
{
  return value[i];
}
```

```
const int& A::operator[](int i)
const {
  return value[i];
}
```

If you have the one to the left you probably want the one to the right, otherwise operator[] cannot be called within const methods.

Example: Dynamic Vector

File "test_vector_01.cpp"

```
#include "iostream"
#include "mdp_vector.h"

Vector<int> square(Vector<int> a) {
  Vector<int> b(a.length());
  for(int i=0; i<a.length(); i++)
    b[i]=a[i]*a[i];
  return b;
}

void main() {
  Vector<int> a,b;
  a.resize(3);
  a[0]=3;
  a[1]=4;
  a[2]=5;
  cout << "a=" << a << endl;
  b=square(a);
  cout << "b=" << b << endl;
}
```

output shell

```
a=[3, 4, 5]
b=[9, 16, 25]
press ENTER to continue...
```

Example: Dynamic Vector

File "mdp_vector.h"

```
Template <class T>
class Vector {

 private:
  int size;
  T*  p;

 public:
  void resize(int i) {
    if(size!=0) delete[] p;
    size=i;
    if(size>0) p=new T[size];
  }

  Vector(int i=0) {
    size=0;
    resize(i);
  }

  ~Vector() {
    resize(0);
  }
```

```
Vector(const Vector& a) {
  size=0;
  resize(a.size);
  for(int i=0; i<size; i++)
    p[i]=a.p[i];
}

Vector& operator=(const Vector& a){
  if(&a==this) return (*this);
  resize(a.size);
  for(int i=0; i<size; i++)
    p[i]=a.p[i];
  return (*this);
}

int length() const {
  return size;
}
```

Example: Dynamic Vector

File "mdp_vector.h" (continue)

```
  T& operator[](int i) {
    if(i<0 || i>=size)
      throw Exception("VectorIndexOutOfBoundsException");
    return p[i];
  }

  const T& operator[](int i) const {
    if(i<0 || i>=size)
      throw Exception("VectorIndexOutOfBoundsException");
    return p[i];
  }

  friend ostream& operator<<(ostream& os, const Vector& a) {
    os << "[";
    if(a.size>0)
      cout << a[0];
    for(int j=1; j<a.size; j++)
      os << ", " << a[j];
    os << "]";
    return os;
  }

}; // end class
```

Example: check phases

Program "check_phases.cpp"

```
void check_phases() {
  const double Pi=3.14159265358979323846264;
  const int N=10;
  int i;
  double phase;

  Array<Complex> psi(N);
  Array<Complex> phi(N);
  Array<Complex> chi(N);

  for(i=0; i<N; i++) {
    phase=2.0*Pi*i/N;
    psi[i]=cos(phase)+I*sin(phase);
    phi[i]=cos(2.0*phase)+I*sin(2.0*phase);
    chi[i]=psi[i]*psi[i]-phi[i];
  }
  cout << chi << endl;
}
```

Test high school trigonometry:

cos(2 a) = cos(a)^2 - sin(a)^2

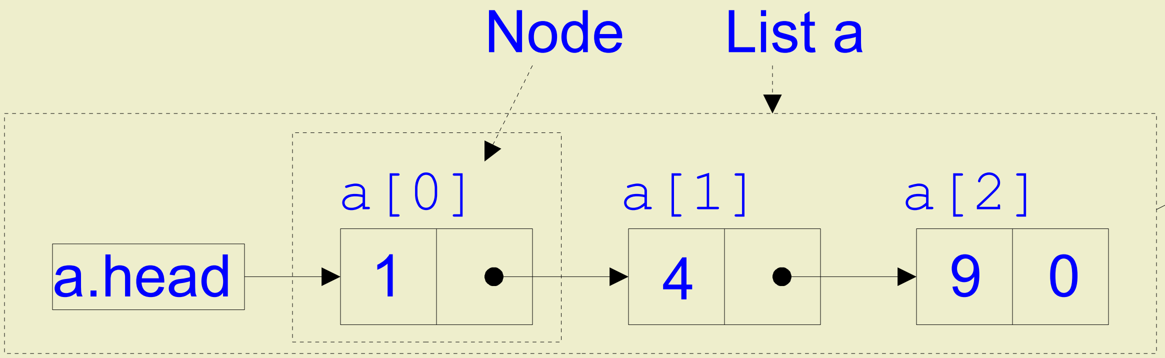
sin(2 a) = 2 sin(a) cos(a)

(in case you did not believe it!)

output shell

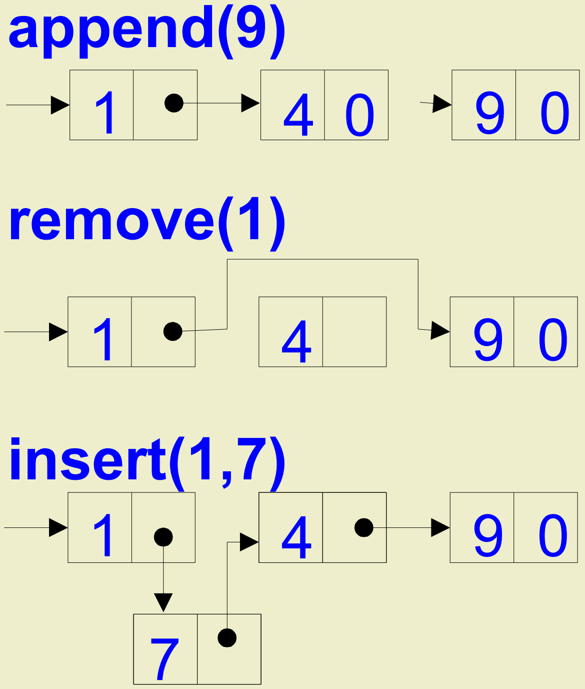
```
[ 0+0*I, 0+0*I, 0+0*I,
0+0*I, 0+0*I, 0+0*I,
0+0*I, 0+0*I, 0+0*I,
0+0*I ]
press ENTER to continue
...
```

Example: Linked List



output shell

```
a=[1, 4, 9]
```



Program "test_list_01.cpp"

```cpp
#include "iostream.h"
#include "mdp_list.h"

void main() {
  List<int> a,b;
  a.append(1);
  a.append(9);
  a.insert(1,4);
  cout << "a=" << a << endl;
  b=a;
  b[1]=5;
  b.remove(2);
  cout << "b=" << b << endl;
  pause();
}
```

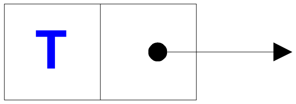
output shell

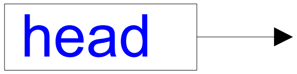
```
a=[1, 4, 9]
b=[1, 5]
press ENTER to continue...
```

Example: Linked List

File "mdp_list.h"

```
template <class T>
class List {
 protected:
  class Node {
   public:
     T     value;
     Node* next;
     Node(T a, Node* b=0) {
      value=a;
      next=b;
    }
  };

 private:
  Node* head;
  int size;
```





```
Public:
  List() { // constructor
   head=0;
   size=0;
 }
 ~List() { erase(); }
 erase() {
   for(int j=size-1; j>=0; j--)
     remove(j);
 }
 List(const List& list) { // c.c.
   head=0;
   size=0;
   for(int i=0;i<list.length();i++)
     append(list[i]);
 }
 List& operator=(const List& list){
   if(&list==this) return (*this);
   erase();
   for(int i=0;i<list.length();i++)
     append(list[i]);
   return (*this);
 }
```

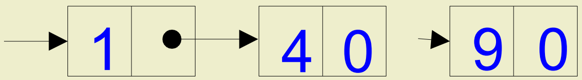
Example: Linked List

File "mdp_list.h" (continue)

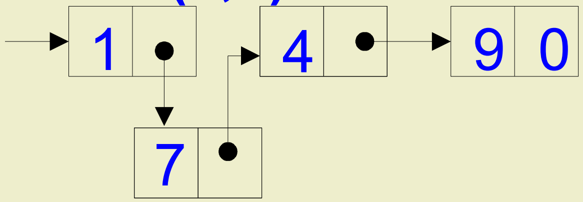
```
void append(T a) {
  if(head==0) {
    head=new Node(a,0);
    size++;
  } else {
    Node* p=head;
    while(p->next!=0) p=p->next;
    p->next=new Node(a,0);
    size++;
  }
}
```

```
void insert(int i, T a) {
  if(i<0 || i>=size) throw
    Exception("Out of bounds");
  if(i==0)
    head=new Node(a,head);
  else {
    Node* p=head;
    for(int j=0;j<i-1;j++)
      p=p->next
    p->next=new Node(a,p->next);
  }
  size++;
}
```

append(9)



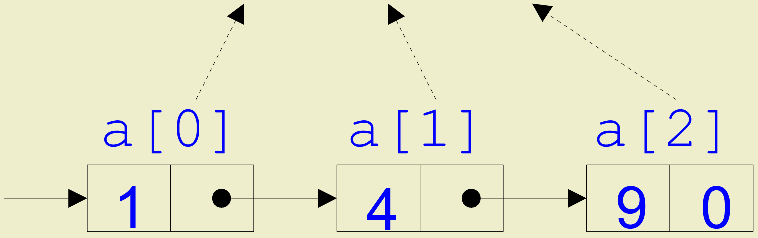
insert(1,7)



Example: Linked List

File "mdp_list.h" (continue)

```
T& operator[](int i) {
  if(i<0 || i>=size)
    throw Exception("ListIndexOutOfBoundsException");
  Node* p=head;
  for(int j=0; j<i; j++) p=p->next;
  return p->value;
}

const T& operator[](int i) const {
  if(i<0 || i>=size
    throw Exception("ListIndexOutOfBoundsException");
  Node* p=head;
  for(int j=0; j<i; j++) p=p->next;
  return p->value;
}
```



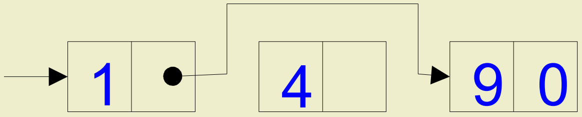
Example: Linked List

File "mdp_list.h" (continue)

```
 void remove(int i) {
  Node* q;
  if(i<0 || i>=size) throw
   Exception("Out of bounds");
  if(i==0) {
   q=head;
   head=head->next;
   delete q;
  } else {
   Node* p=head;
   for(int j=0;j<i-1;j++)
    p=p->next;
   q=p->next;
   p->next=q->next;
   delete q;
  }
  size--;
 }
```

```
 int length() const {
  return size;
 }

 friend ostream& operator<<
 (ostream& os, const List& list){
  os << "[";
  if(list.size>0)
   os << list[0];
  for(int j=1; j<list.size; j++)
   os << ", " << list[j];
  os << "]";
  return os;
 }
}; // end class
```

remove(1)



output shell

```
[1, 4, 9]
```

Example: remove newlines

Program "test_replace_string.h"

```
void replace_string() {
  int i,j,k;
  string in, out, filename, line;
  List<String> a;
  ifstream file;
  cout << "Insert a file name   :"; cin >> filename;
  cout << "String to be replaced:"; cin >> in;
  cout << "To be replaced with  :"; cin >> out;
  file.open(filename.c_str());
  while(true) {
    file >> line; if(file.fail()) break;
    a.append(line);
  }
  for(i=0; i<a.length(); i++) {
    for(k=0;
        k<a[i].length() && (j=a[i].find(in,k))>=0;
        k=k+j+out.length())
        a[i]=a[i].replace(j,in.length(),out);
    cout << a[i] << endl;
  }
  file.close();
}
```

Example: QuickSort

File "mdp_algorithms.h"

```
template<class T>
void QuickSort(T& A, int p, int r) {
  int i,j,q;
  if(p<r) {
    i=p-1;
    j=r+1;
   while(true) {
     for(i++;A[i]<A[p];i++);
     for(j--;A[j]>A[p];j--);
     if(i<j) Swap(A[i],A[j]);
     else { q=j; break; }
   }
    InsertionSort(A,p,q);
    InsertionSort(A,q+1,r);
  }
}

template<class T>
void QuickSort(T& A) {
  QuickSort(A,0,A.length()-1);
}
```

Program "test_sort.h"

```
void sort_string() {
  String s;
  cout << "Input  string:";
  cin >> s;
  InsertionSort(s);
  cout << "Sorted string :"
       << s << endl;
}
```

output shell

```
0872936451
0123456789
press ENTER to continue...
```

Example: QuickSort

Program "test_sort.h"

```
void sort_vector_of_int() {
  int i;
  cout << "Elements of the vector (1-10):"; cin >> i;
  cout << endl;
  Vector<int> a(i);
  for(i=0;i<a.length(); i++)
    cout << "a[" << i<< "]="; cin >> a[i];

  InsertionSort(a);

  cout << "Sorted vector:\n";
  for(i=0;i<a.length(); i++)
    cout << "a[" << i<< "]=" << a[i] << endl;
}
```

Example: QuickSort

Program "test_sort.h"

```
void sort_vector_of_string() {
  int i;
  cout << "Elements of the vector (1-10):";
  cin >> i;
  Vector<String> a(i);
  for(i=0; i<a.length(); i++)
    cout << "a[" << i << "]="; cin >> a[i];

  InsertionSort(a);

  cout << "Sorted vector:\n";
  for(i=0;i<a.length(); i++)
    cout << "a[" << i<< "]=" << a[i] << endl;
}
```

Example: QuickSort

Program "test_sort.h"

```
void sort_list_of_string() {
  int i;
  String input;
  cout << "Insert array elements [ENTER] to finish";
  List<String> a;
  for(i=0;; i++)
    cout << "a[" << i << "]="; cin >> input;
    if(input!="") a.append(input); else break;

  InsertionSort(a);

  cout << "Sorted list:\n";
  for(i=0;i<a.length(); i++)
    cout << "a[" << i<< "]=" << a[i] << endl;
}
```

Week 6

MIDTERM

Inheritance
(class Map)

Inheritance

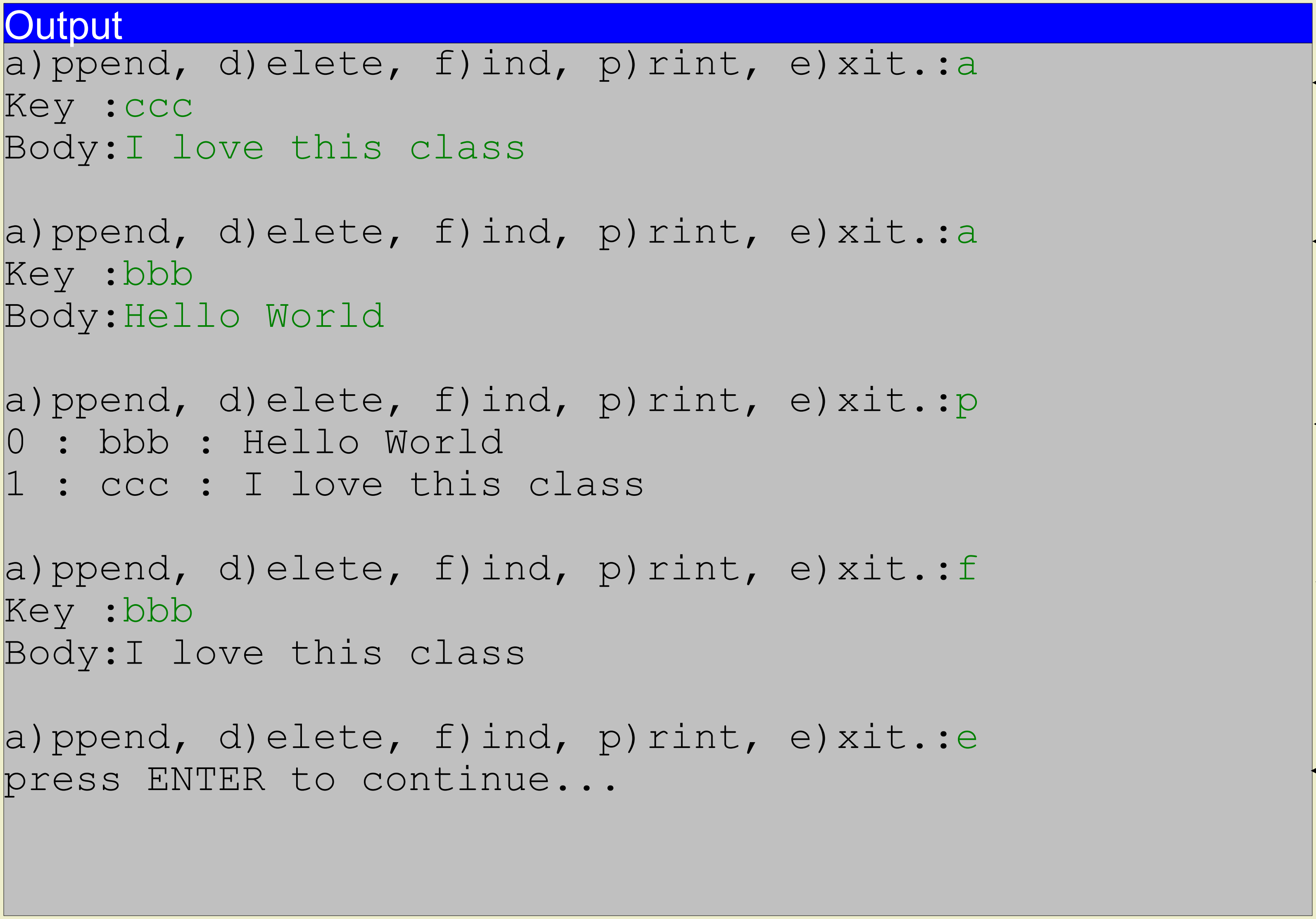
Java Program

```
public class BC {
  private int i;
  public int get() {
    return i;
  }
}

public class DC extends BC {
  public int getTwice() {
    return 2*get();
  }
}
```

Program "inheritance_01.cpp"

```
class BC {
  private: int i;
  public: int get() {
    return i;
  }
};

class DC : public BC {
  public: int getTwice() {
    return 2*get();
  }
};
```

Example: Map



Output

```
a)ppend, d)elete, f)ind, p)rint, e)xit.:a
Key :ccc
Body:I love this class

a)ppend, d)elete, f)ind, p)rint, e)xit.:a
Key :bbb
Body:Hello World

a)ppend, d)elete, f)ind, p)rint, e)xit.:p
0 : bbb : Hello World
1 : ccc : I love this class

a)ppend, d)elete, f)ind, p)rint, e)xit.:f
Key :bbb
Body:I love this class

a)ppend, d)elete, f)ind, p)rint, e)xit.:e
press ENTER to continue...
```

append record and sort

append record and sort

print all records

find record by key

Exit

Example: Map

Program "mdp_map.h"

```
Void main_map() {
  int i;
  String choice, key, body;
  Map<String,String> db;
  while(true) {
    cout << "\na)ppend, d)elete, f)ind, p)rint, e)xit.:";
    cin >> choice;
    switch(choice[0]) {
    case 'a': cout << "Key :"; cin >> key;
              cout << "Body:"; cin >> body;
              db.appendRecord(key,body); break;
    case 'd': cout << "Key :"; cin >> key;
              if(db.hasKey(key)) db.deleteRecord(key); break;
    case 'f': cout << "Key :"; cin >> key;
              if(db.hasKey(key)) cout << "Body:" << db(key) << endl;
              break;
    case 'p': for(i=0; i<db.length(); i++)
                cout << i << " : " << db[i].key
                          << " : " << db[i].body << endl;
              break;
    case 'e': return;
    }
  }
}
```

Example: Map

File "mdp_map.h"

```
Template<class S, class T>
class Record {
public:
  S key;
  T body;
  Record() {}
  Record(const S&  s, const T& t) {
   key=s;
   body=t;
  }
  friend bool operator<(const Record& a,
                        const Record& b) {
    return (a.key<b.key);
  }
  friend bool operator>(const Record& a,
                        const Record& b) {
    return (a.key>b.key);
  }
};
```

Example: Map

File "mdp_map.h" (continued)

```
Template<class S, class T>
class Map : public List<Record<S,T> > {

public:
  Map() { } // very important!!!
  int recordIndex(const S& key) const {
    for(int i=0; i<length(); i++)
     if((*this)[i].key==key) return i;
   return -1;
  }

  bool hasKey(const S& key) const {
    if(recordIndex(key)<0) return false;
   return true;
  }

  void appendRecord(const S& key, const T& body) {
    append(Record<S,T>(key,body));
    InsertionSort(*this);
  }
```

Example: Map

File "mdp_map..h" (continued)

```
Public:
 bool deleteRecord(const S& key) {
   int i=recordIndex(key);
    if(i<0) return false;
   remove(i);
   return true;
 }

 T& operator()(const S& key) {
    int i=recordIndex(key);
   if(i<0) throw Exception("MapIndexOutOfBounds");
   return (*this)[i].body;
 }

 const T& operator()(const S& key) const {
    int i=recordIndex(key);
   if(i<0) throw Exception("MapIndexOutOfBounds");
   return (*this)[i].body;
 }
```

Example: Map

File "mdp_map.h" (continued)

```
bool save(string filename) {
  ofstream file;
  file.open(filename.c_str());
  for(int i=0; i<length(); i++) {
   file << "RECORD N. " << i << endl;
   file << (*this)[i].key  << endl;
   file << (*this)[i].body << endl;
 }
 file.close();
 return true;
}
```

Example: Map

File "mdp_map.h" (continued)

```
  bool load(String filename) {
    String dummy;
    S key;
    T body;
    ifstream file;
    file.open(filename.c_str());
    if(!file) return false;
    while(true) {
      file >> dummy;
      if(file.fail()) break;
      file >> key;
      file >> body;
      appendRecord(key,body);
    }
    file.close();
    return true;
  }
}; // end class
```

Inheritance, Interfaces and Polymorphism

Polymorphism: overloading virtual methods

Inheritance and polymorphism

Program "inheritance_02.cpp"

```
#include "iostream"
class BC {
 public:
  void f() {
   cout << "BC::f()\n";
  }
};
class DC : public BC {
 public:
  void f() {
   cout << "DC::f()\n";
  }
};
void main() {
  BC b;
  DC d;
  b.f();
  d.f();
}
```

output shell

```
BC::f()
DC::f()
press ENTER to continue ...
```

Program "inheritance_03.cpp"

```
#include "iostream"
class BC {
 public:
  void f() {
   cout << "BC::f()\n";
  }
};
class DC : public BC {
 public:
  void f() {
    cout << "DC::f()\n";
  }
};
void main() {
  BC* b1=new BC;
  BC* b2=new DC;
  b1->f();
  b2->f();
}
```

output shell

```
BC::f()
BC::f()
press ENTER to continue ...
```

Inheritance: constructors and destructors

Program "inheritance_03.cpp"

```
#include "iostream"

class BC {
 public:
  BC() {
    cout << "BC constructor\n";
  }
};
class DC : public BC {
 public:
  DC() {
    cout << "DC constructor\n";
  }
};

void main() {
  DC d;
}
```

output shell

```
BC constructor
DC constructor
press ENTER to continue ...
```

Program "inheritance_04.cpp"

```
#include "iostream"

class BC {
 public:
  ~BC() {
    cout << "BC destructor\n";
  }
};
class DC : public BC {
 public:
  ~DC() {
    cout << "DC destructor\n";
  }
};

void main() {
  DC d;
}
```

output shell

```
DC destructor
BC destructor
press ENTER to continue ...
```

Inheritance: constructor warning

Program "inheritance_05.cpp"

dangerous

```
#include "iostream"

class BC {
 public:
  BC() {
    cout << "BC constructor\n";
  }
};
class DC : public BC {
 public:
  DC(int n) {
    cout << "DC constructor\n";
  }
};

void main() {
  DC d;
}
```

fix

output shell

```
compiler error
```

Program "inheritance_06.cpp"

```
#include "iostream"

class BC {
 public:
  BC() {
    cout << "BC constructor\n";
  }
};
class DC : public BC {
 public:
  DC(int n) : BC() {
    cout << "DC constructor\n";
  }
};

void main() {
  DC d;
}
```

output shell

```
BC constructor
DC constructor
press ENTER to continue ...
```

Inheritance and Polymorphism: virtual functions

Program "virtual_01.cpp"

dangerous

```
#include "iostream"
class BC {
 public:
  void speak() {
    cout << "BC speaks\n";
  }
};
class DC : public BC {
 public:
  void speak() {
    cout << "DC speaks\n";
  }
};
void main() {
  BC *d=new DC();
  (*d).speak();
  delete d;
}
```

output shell

```
BC speaks
press ENTER to continue ...
```

fix

Program "virtual_02.cpp"

```
#include "iostream"
class BC {
 public:
  virtual void speak() {
    cout << "BC speaks\n";
  }
};
class DC : public BC {
 public:
  void speak() {
    cout << "DC speaks\n";
  }
};
void main() {
  BC *d=new DC();
  (*d).speak();
  delete d;
}
```

output shell

```
DC speaks
press ENTER to continue ...
```

Inheritance and Polymorphism: destructor warning

Program "virtual 03.cpp"

dangerous

```
#include "iostream"
class BC {
 public:
  ~BC() {
     cout << "BC destructor\n";
  }
};
class DC : public BC {
 public:
  ~DC() {
     cout << "DC destructor\n";
  }
};

void main() {
  BC* d=new DC();
  delete d;
}
```

output shell

```
BC destructor
press ENTER to continue ...
```

fix

Program "virtual 04.cpp"

```
#include "iostream"
class BC {
 public:
  virtual ~BC() {
     cout << "BC destructor\n";
  }
};
class DC : public BC {
 public:
  ~DC() {
     cout << "DC destructor\n";
  }
};

void main() {
  BC* d=new DC();
  delete d;
}
```

output shell

```
DC destructor
BC destructor
press ENTER to continue ...
```

Java vs C++
virtual methods

Java methods are all virtual by default
(because Java does not distinguish compile-time binding vs run-time binding)

C++ methods are all non-virtual by default
(because C++ prefers compile-time binding when possible)

If you understand inheritance in Java and want imitate it,
declare all your C++ methods virtual...

...except the constructors which cannot be virtual!

Tip: declare the destructor always virtual.

Inheritance and Polymorphism: name hiding

Program "virtual_05.cpp"

wrong

```
#include "iostream"
class BC {
 public:
  void m(int i) {
    cout << "m(int)\n";
  }
};

class DC : public BC {
 public:
  void m() {
    cout << "m()\n";
  }
};

void main() {
  DC d;
  d.m(3); // not defined
}
```

Fix

Program "virtual_06.cpp"

```
#include "iostream"
class BC {
 public:
  void m(int i) {
    cout << "m(int)\n";
  }
};

class DC : public BC {
 public:
  void m() {
    cout << "m()\n";
  }
  void m(int i) { BC::m(i); };
};
void main() {
  DC d;
  d.m(3); // OK here
}
```

output shell

```
m(int)
press ENTER to continue ...
```

derived method m() **hides** all inherited methods with same name

Interface and Polymorphism

Java Program

```
public interface IC {
  public int get();
}

public class DC implements IC {
  private int i;
  public int get() {
    return i;
  }
}
```

Program "interface_01.cpp"

```
class IC {
  public: virtual int get()=0;
};

class DC : public IC {
  private: int i;
  public: int get() {
    return i;
  }
};
```

An interface class (IC) in C++ is an ordinary class which methods are all purely virtual (**`virtual ... =0`**). Such a class is called Abstract Base Class.

Interface and Polymorphism

Program "interface_02.cpp"

```
#include "iostream"
class Widget {
 public:
  virtual void show() { cout << "I, Widget\n"; };
};
class Square : public Widget {
 public:
  virtual void show() { cout << "I, Square\n"; };
};
class Circle : public Widget {
 public:
  virtual void show() { cout << "I, Circle\n"; };
};

void main() {
  Widget* w=(Widget*) new Square;
   (*w).show();
  delete w;
}
```

w can be Widget, Square or Circle

output shell

```
I, Square
press ENTER to continue ...
```

Interface and Polymorphism

Program "interface_03.cpp"

```
#include "iostream"
class Widget {
 public:
  virtual void show()=0;
};
class Square : public Widget {
 public:
  virtual void show() { cout << "I, Square\n"; };
};
class Circle : public Widget {
 public:
  virtual void show() { cout << "I, Circle\n"; };
};

void main() {
  Widget* w=(Widget*) new Square;
  (*w).show();
  delete w;
}
```

Abstract Base Calss (Interface Class)

must have show()

w can be Square or Circle but not Widget

output shell

```
I, Square
press ENTER to continue ...
```

Example: simplified Blackjack

Deck contains 40 cards (4 colors and 10 values: 1,2,3,4,5,6,7,8,9 and 10)

Cards are shuffled.

Each player (including Dealer) gets two cards.

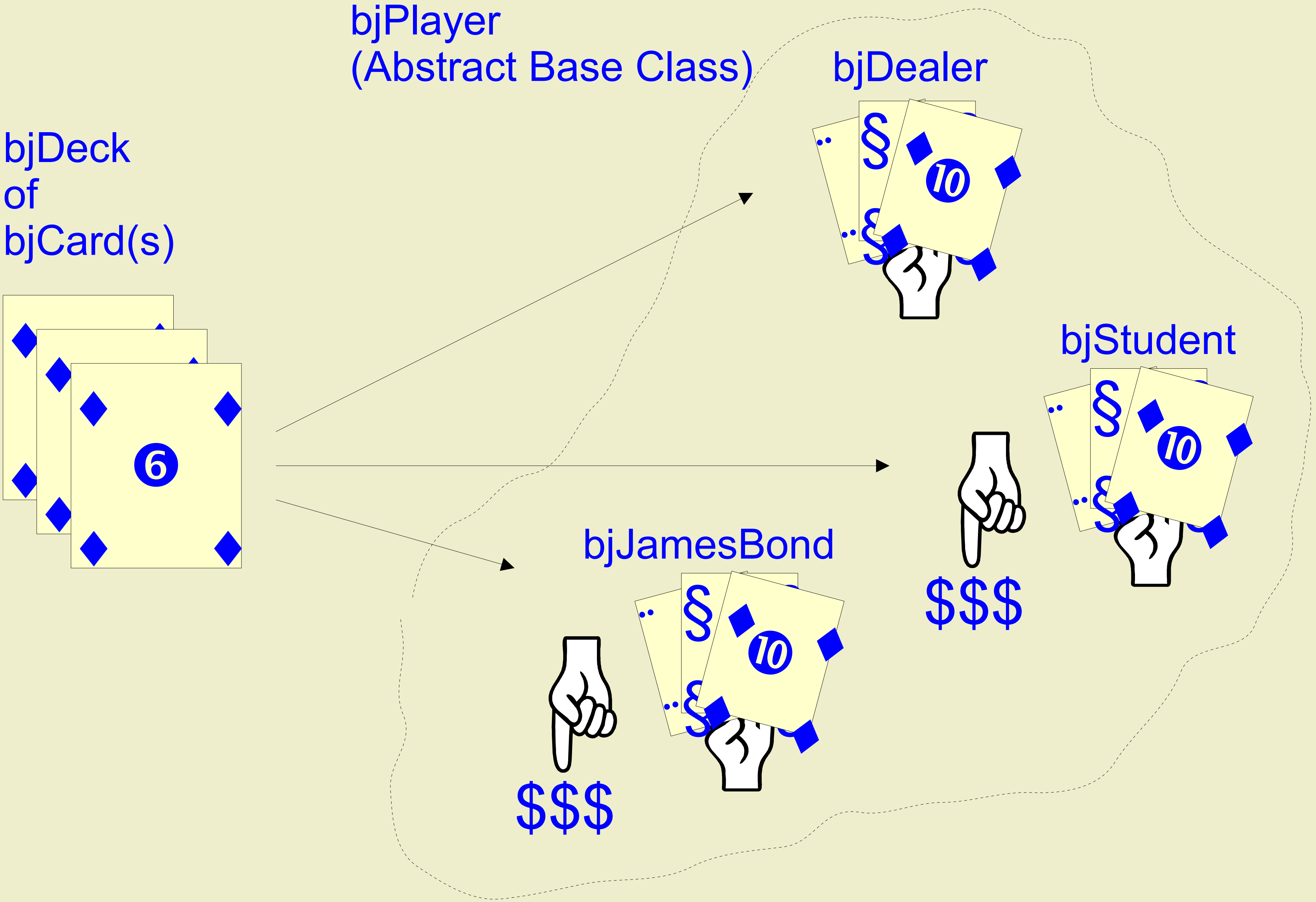
Each player applies **his/her own strategy:**
makes a bet against and asks for one or more card(s).

If a player gets a score closer to 21 than the Dealer,
the Dealer pays player.

If a player exceeds 21 he/she pays the Dealer.

If the Dealer exceeds 21 he pays all players that did not exceed
21 with their relative bets.

Example: simplified Blackjack



bjPlayer
(Abstract Base Class)

bjDealer

bjDeck
of
bjCard(s)

bjStudent

bjJamesBond

$$$

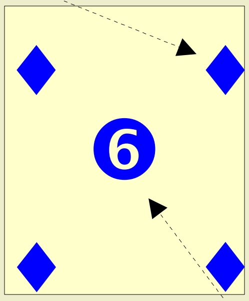
$$$

Example: simplified Blackjack

File "mdp_blackjack.h"

```
class bjCard {
public:
  int c;
  int v;
  int color() const {
    return c;
  }
  int value() const {
    return v;
  }
  bjCard() { }
  bjCard(int cc, int vv) {
    c=cc;
    v=vv;
  }
};
```

Color=♣♦♥♠

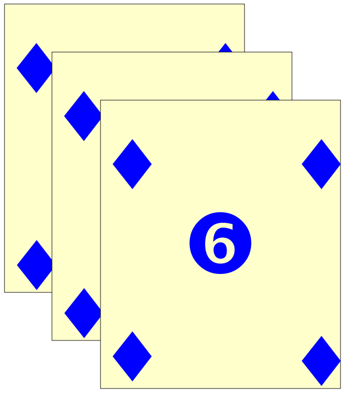


Value=❶❷❸❹❺❻❼❽❾❿

Example: simplified Blackjack

File "mdp_blackjack.h" (continue)

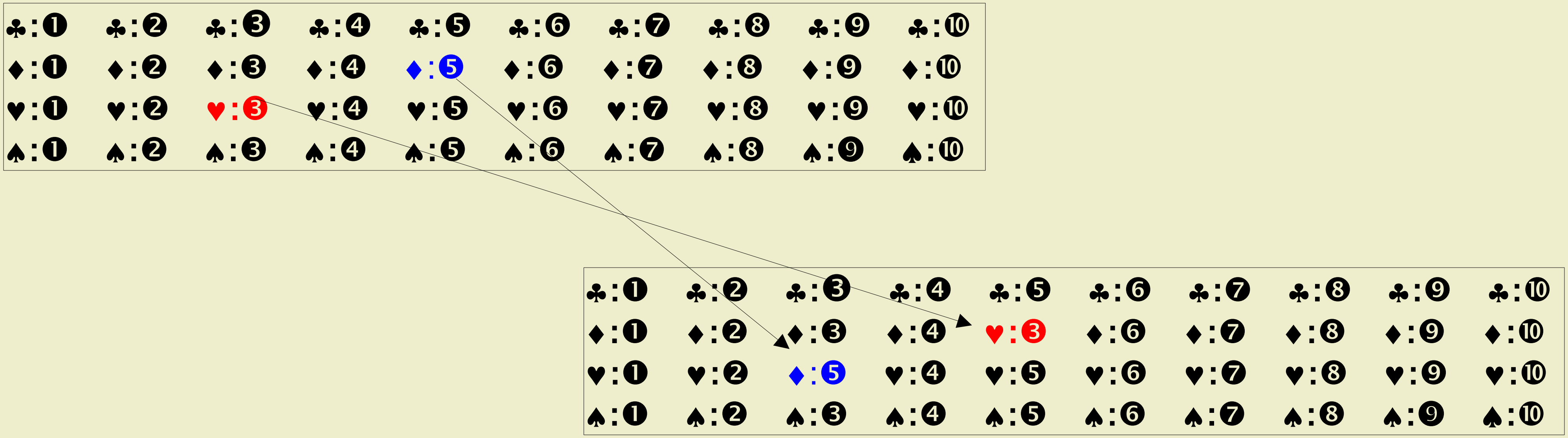
```
class bjDeck {
public:
  enum {minColor=0, maxColor=3, minValue=1, maxValue=10 };
  List<bjCard> cards;
  bjDeck() {
   int color, value;
   for(color=minColor; color<=maxColor; color++)
      for(value=minValue; value<=maxValue; value++)
        cards.append(bjCard(color,value));
  }
  int remainingCards() const {
    return cards.length();
  }
  bjCard getCard() {
     if(cards.length()==0)
       throw Exception("bjDeckEmptyExcpetion");
     bjCard topcard=cards[0];
    cards.remove(0);
    return topcard;
  }
```



Example: simplified Blackjack

File "mdp_blackjack.h" (continue)

```
  void shuffle(int n=2) {
    int i,j,k;
    for(k=0; k<n; k++) {
       for(i=0; i<cards.length(); i++) {
         // function rand() requires #include "stdlib.h"
         j=rand() % cards.length();
         Swap(cards[i],cards[j]);
       }
    }
  }
};
```



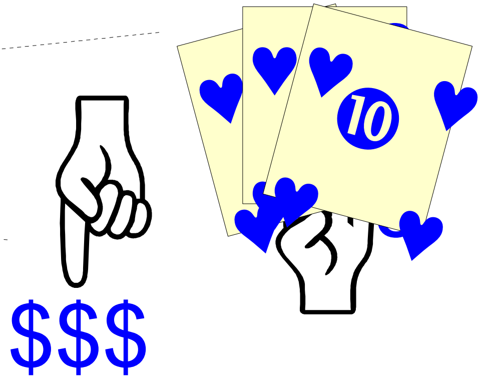
♣:❶	♣:❷	♣:❸	♣:❹	♣:❺	♣:❻	♣:❼	♣:❽	♣:❾	♣:❿
♦:❶	♦:❷	♦:❸	♦:❹	♦:❺	♦:❻	♦:❼	♦:❽	♦:❾	♦:❿
♥:❶	♥:❷	♥:❸	♥:❹	♥:❺	♥:❻	♥:❼	♥:❽	♥:❾	♥:❿
♠:❶	♠:❷	♠:❸	♠:❹	♠:❺	♠:❻	♠:❼	♠:❽	♠:❾	♠:❿

♣:❶	♣:❷	♣:❸	♣:❹	♣:❺	♣:❻	♣:❼	♣:❽	♣:❾	♣:❿
♦:❶	♦:❷	♦:❸	♦:❹	♥:❸	♦:❻	♦:❼	♦:❽	♦:❾	♦:❿
♥:❶	♥:❷	♦:❺	♥:❹	♥:❺	♥:❻	♥:❼	♥:❽	♥:❾	♥:❿
♠:❶	♠:❷	♠:❸	♠:❹	♠:❺	♠:❻	♠:❼	♠:❽	♠:❾	♠:❿

Example: simplified Blackjack

File "mdp_blackjack.h" (continue)

```
class bjPlayer {
public:
  string name;
  List<bjCard> hand;
  int bet;
  int portfolio;
  bjPlayer() { portfolio=0; handReset(); }
  void handReset() {
    bet=0;
    hand.erase();
  }
  void askCard(bjDeck& deck) {
    hand.append(deck.getCard());
  }
  int handValue() const {
    int value=0;
   for(int i=0; i<hand.length(); i++)
      value=value+hand[i].value();
   return value;
  }
  virtual void play(bjDeck&)=0;
};
```



$$
$$$
$$$$

game strategy

Example: simplified Blackjack

File "mdp_blackjack.h" (continue)

```
class bjDealer : public bjPlayer {
public:
   void play(bjDeck& deck) {
     bet=100;
     while(handValue()<17) askCard(deck);
   }
};
```

game strategy

Example: simplified Blackjack

File "mdp_blackjack.h" (continue)

```
class bjStudent : public bjPlayer {
public:
  void play(bjDeck& deck)  {
    bet=100;
    while(handValue()<15)  {
       bet=bet+100;
       askCard(deck);
    }
  }
};
```

game strategy

Example: simplified Blackjack

File "mdp_blackjack.h" (continue)

```
class bjJamesBond : public bjPlayer {
public:
  void play(bjDeck& deck) {
    bet=100;
    while(handValue()<19) {
       bet=2*bet;
       askCard(deck);
    }
  }
};
```

game strategy

Example: simplified Blackjack

Program "mdp_blackjack.h" (continue)

```
void play_blackjack() {
  bjDeck fulldeck, deck;
  List<bjPlayer*> players;
  int i,match, nmatches=100;

  // selecet the players
  players.append((bjPlayer*) new bjDealer);
  players.append((bjPlayer*) new bjStudent);
  players.append((bjPlayer*) new bjJamesBond);
  // append more if you like ...

  for(match=0; match<nmatches; match++) {
    deck=fulldeck;
    deck.shuffle();
    // each player asks for two cards and plays
    for(i=0; i<players.length(); i++) {
      players[i]->handReset();
      players[i]->askCard(deck);
      players[i]->askCard(deck);
      players[i]->play(deck);
    }
```

Example: simplified Blackjack

Program "mdp_blackjack.h" (continue)

```
    // settle bets
    for(i=1; i<players.length(); i++) {
       if(players[i]->handValue()<=21 &&
          players[i]->handValue()>players[0]->handValue()) {
         players[i]->portfolio+=players[i]->bet;
         players[0]->portfolio-=players[i]->bet;
       } else {
         players[i]->portfolio-=players[i]->bet;
         players[0]->portfolio+=players[i]->bet;
       }
    }
    // print outcome of the match
    cout << "MATCH N. " << match << endl;
    for(i=0; i<players.length(); i++) {
      cout << "  player: " << i
           << ", bet: " << players[i]->bet
           << ", hand:" << players[i]->handValue()
           << ", portfolio: " << players[i]->portfolio << endl;
    }
  } // for ... match ...

  // deallocate players ...
}
```

Example: simplified Blackjack

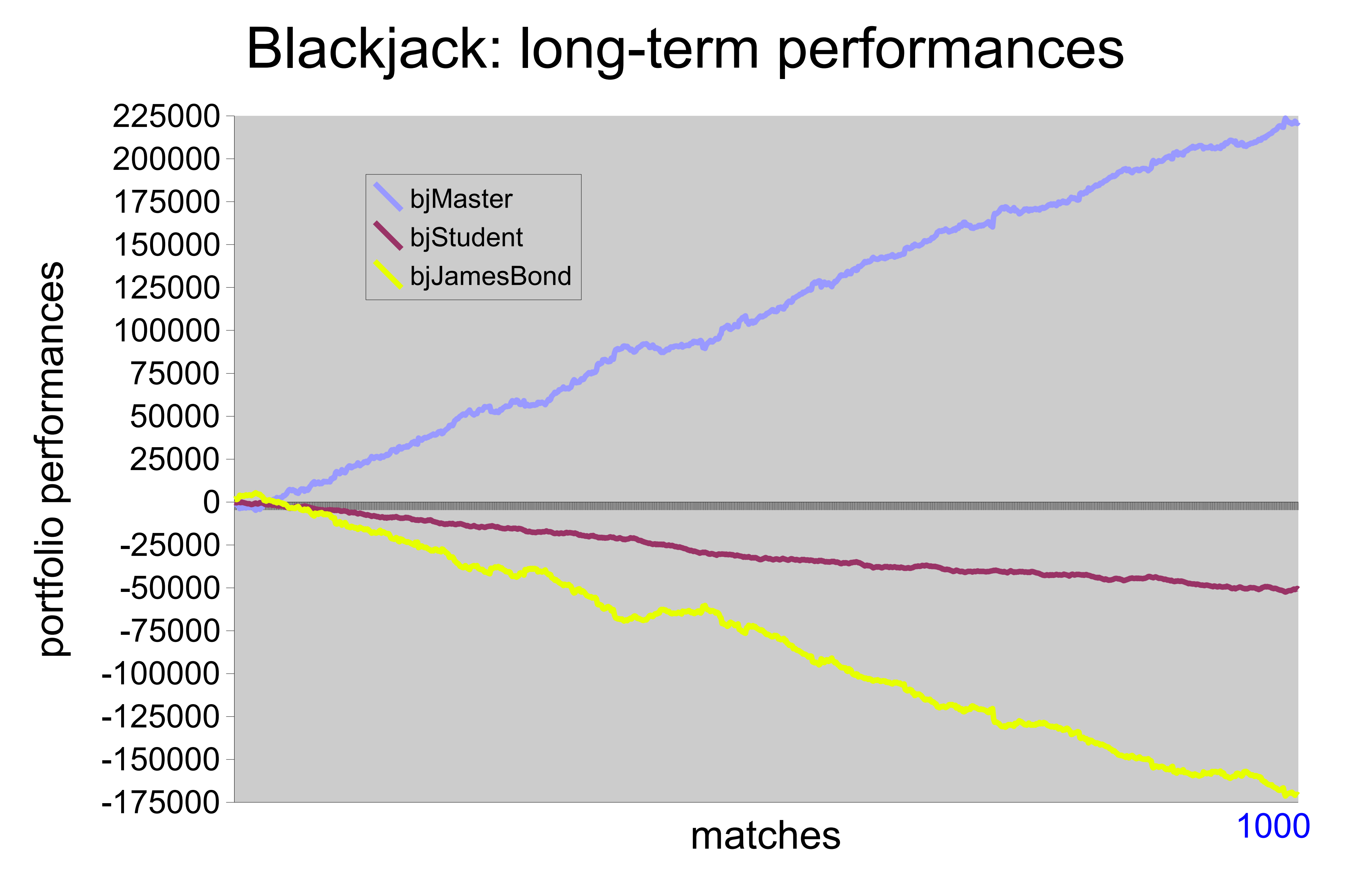
Simulate different Blackjack strategies

Output

```
MATCH N. 0
  player: 0, bet: 0,   hand:22, portfolio: 3500
  player: 1, bet: 300, hand:15, portfolio: -300
  player: 2, bet: 3200,hand:23, portfolio: -3200
MATCH N. 1
  player: 0, bet: 0,   hand:22, portfolio: 3900
  player: 1, bet: 200, hand:15, portfolio: -500
  player: 2, bet: 200, hand:21, portfolio: -3400
MATCH N. 2
  player: 0, bet: 0,   hand:19, portfolio: 3400
  player: 1, bet: 300, hand:20, portfolio: -200
  player: 2, bet: 200, hand:20, portfolio: -3200
...
MATCH N. 99
  player: 0, bet: 0,   hand:20, portfolio: 48400
  player: 1, bet: 200, hand:17, portfolio: -19600
  player: 2, bet: 400, hand:24, portfolio: -28800

press ENTER to continue...
```

player 0 always wins on the long run!

Example: simplified Blackjack



Example: simplified Blackjack interactive game

File "mdp_blackjack.h" (continue)

```
class bjInteractive : public bjPlayer {
public:
  void play(bjDeck& deck) {
   int i, b, c;
   bet=0;
   cout << "\nYour turn " << name << endl;
    while(handValue()<21) {
      cout << "You have the following cards:\n";
      for(i=0; i<hand.length(); i++)
        cout << hand[i].value() << " ";
      cout << "\nHow much do you want to bet? ";
      cin >> b;
      bet=bet+b;
      cout << "Your total bet is " << bet << endl;
      cout << "Do you want a card (0 - no, 1, yes)?";
      cin >> c;
      if(c==1) askCard(deck); else break;
    }
  }
};
```

Example: simplified Blackjack interactive game

Program "mdp_blackjack.h" (continue)

```
void play_blackjack() {
  bjDeck fulldeck, deck;
  List<bjPlayer*> players;
  int i,match, nmatches=100;

  // selecet the players
  players.append((bjPlayer*) new bjDealer);
  players.append((bjPlayer*) new bjStudent);
  players.append((bjPlayer*) new bjJamesBond);
  players.append((bjPlayer*) new bjInteractive);

  players[3]->name="Massimo";

  for(match=0; match<nmatches; match++) {
    deck=fulldeck;
    deck.shuffle();
    // each player asks for two cards and plays
    for(i=0; i<players.length(); i++) {
      players[i]->handReset();
      players[i]->askCard(deck);
      players[i]->askCard(deck);
      players[i]->play(deck);
    }
```

Example: simplified Blackjack interactive game

Playing against the virtual players

Output

```
Your turn Massimo
You have the following cards:
5 8
How much do you want to bet? 1000
Your total bet is 1000
Do you want a card (0 - no, 1, yes)?1
You have the following cards:
5 8 1
How much do you want to bet? 1000
Your total bet is 2000
Do you want a card (0 - no, 1, yes)?0
MATCH N. 0
  player: 0 , bet: 100, hand:19, portfolio: 700
  player: 1 , bet: 300, hand:15, portfolio: -300
  player: 2 , bet: 1600, hand:21, portfolio: 1600     ◄.........
  player: 3 Massimo, bet: 2000, hand:14, portfolio: -2000

...
press ENTER to continue...
```

STL

Our classes	**STL classes**
String	string
Vector<T>	vector<T>
List<T>	list<T>
Map<S,T>	map<S,T>
Advantages:	**Advantages:**
Portable (ANSI C++)	Commonly used
Safe, use exceptions	Faster, Optimized for speed
No need for iterators	Extensive libraries and docs
Database sorts elements	
Disadvantages:	**Disadvantages:**
Not optimized for speed	Different implementation may vary
Not many methods implemented	Use of exception not guaranteed
	map does not sort elements
	functions sort requires use of iterators