PANORAMICA DEL C++

Roberto Nardone, Luigi Romano

ROADMAP

- Cos'è il C++?
- Paradigmi di programmazione
- Programmazione Procedurale
- Programmazione Modulare
- Astrazione Dati
- Programmazione Orientata agli Oggetti
- Programmazione Generica

COS'È IL C++?

- C++ è un linguaggio di programmazione *general purpose* che
  - è "migliore" del C
  - supporta l'astrazione dei dati
  - supporta la programmazione orientata agli oggetti
  - supporta la programmazione generica

- Un linguaggio supporta un *paradigma di programmazione* se fornisce una serie di funzionalità che rendono conveniente usare quello stile e che
  - siano elegantemente integrate nel linguaggio
  - possono essere usate simultaneamente
  - siano principalmente *general purpose*
  - non devono imporre un *overhead* significativo ai programmi che non le richiedono

- Un utente dovrebbe conoscere solo il sottoinsieme del linguaggio esplicitamente usato per scrivere il programma

- Si basa sulla suddivisione di algoritmi che risolvano un dato problema in *funzioni/procedure*
- Una *funzione* è una porzione di codice che esegue un compito ed è indipendente dal resto del codice
- Sono disponibili tecniche per il *passaggio di argomenti* e la restituzione di valori dalle funzioni

```
double sqrt(double arg)          // funzione: arg è l'argomento, double il tipo ritornato
{
        // code for calculating a square root
}

void f()                         // procedura: nessun argomento
{
        double root2 = sqrt(2);
        // ...
}
```

Ogni nome ed ogni espressione ha un tipo che determina le operazioni che possono essere eseguite

```
int inch;          // dichiarazione di un nome variabile di tipo intero
bool bit;          // dichiarazione di un nome variabile di tipo booleano
char letter;       // dichiarazione di un nome variabile di tipo carattere
double number;     // dichiarazione di un nome variabile di tipo floating-point
```

Una dichiarazione è un'istruzione che introduce un nome in un programma

Operatori aritmetici

```
+    // addizione, sia unaria che binaria
-    // sottrazione, sia unaria che binaria
*    // moltiplicazione
/    // divisione
%    // resto
```

Operatori di confronto

```
==    // uguale
!=    // non uguale
<     // minore
>     // maggiore
<=    // minore o uguale
>=    // maggiore o uguale
```

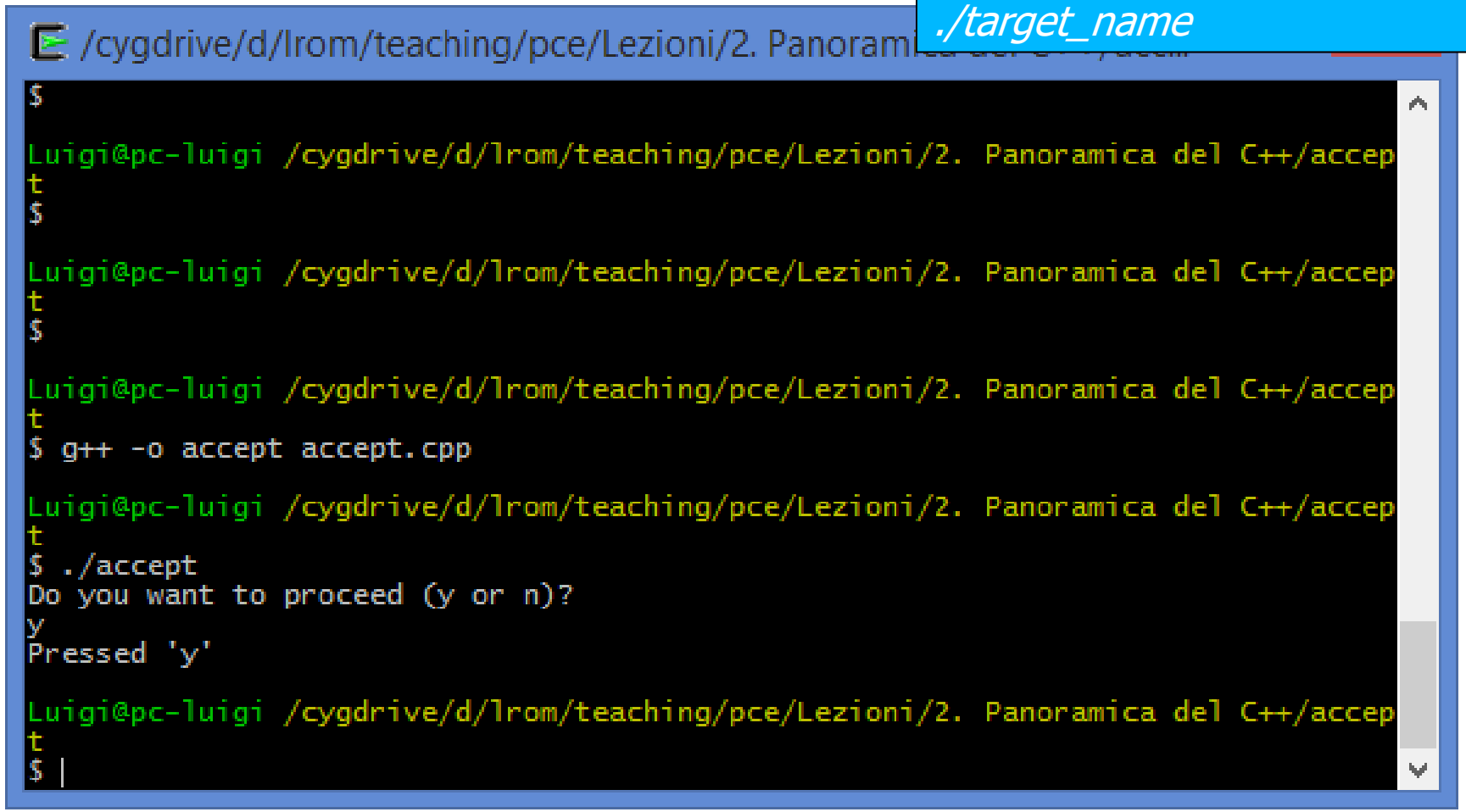
cout: istruzione di visualizzazione a video

cin: istruzione di lettura di una stringa da tastiera

```
bool accept()
{
        cout<<"Do you want to proceed (y or n)?\n";      // write question
        char answer = 0 ;
        cin>>answer;                                     // read answer
        if(answer =='y') return true;
        return false;
}
```

ACCEPT (): COMPILAZIONE ED ESECUZIONE

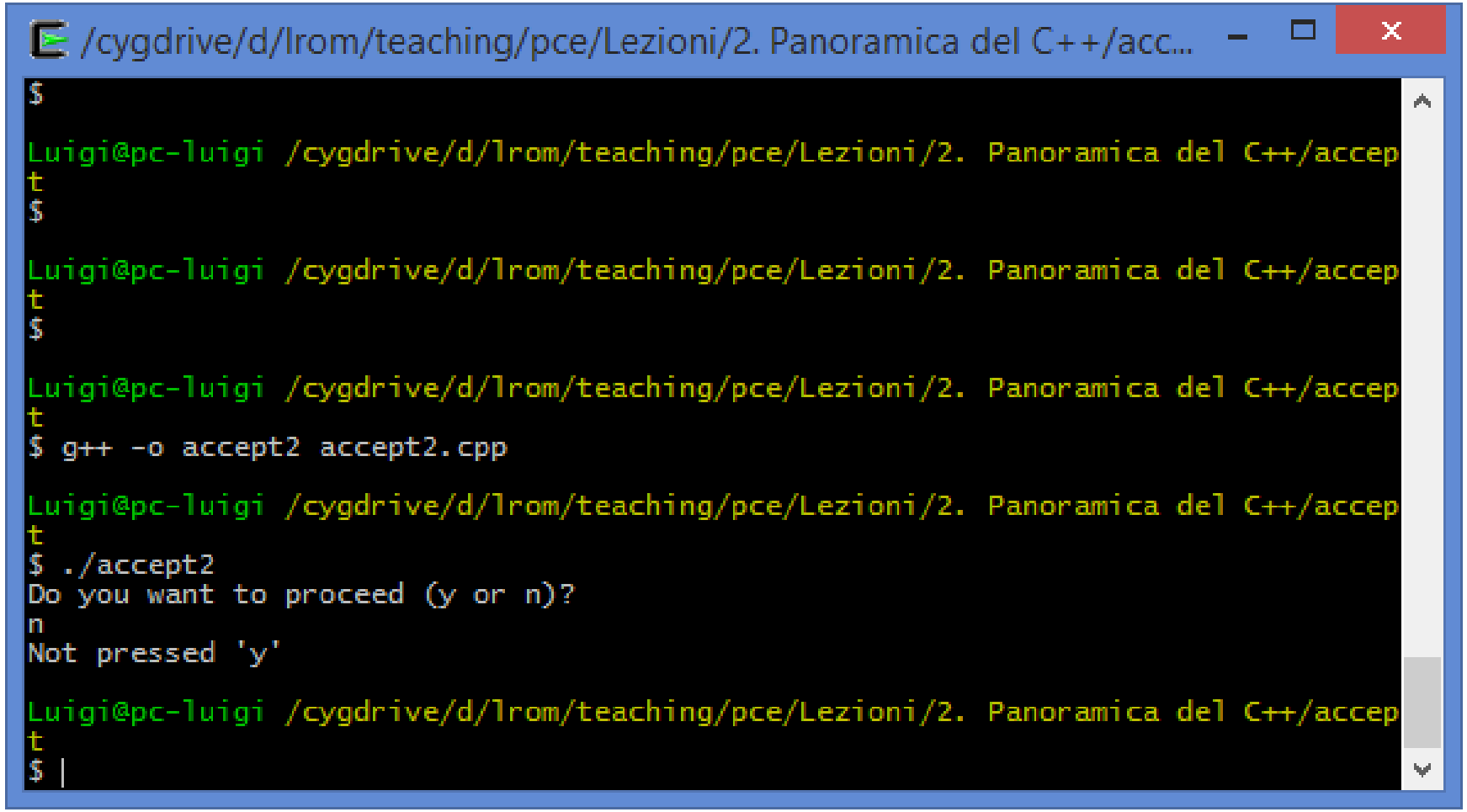
g++ -o target_name file_name
./target_name



switch: confronta un valore con una serie di costanti

```
bool accept2()
{
        cout<<"Do you want to proceed (y or n)?\n";        // write question
        char answer = 0 ;
        cin>>answer;                                       // read answer
        switch(answer) {
                case 'y':
                        return true;
                case 'n':
                        return false;
                default:
                        cout<<"I'll take that for a no.\n";
                        return false;
        }
}
```

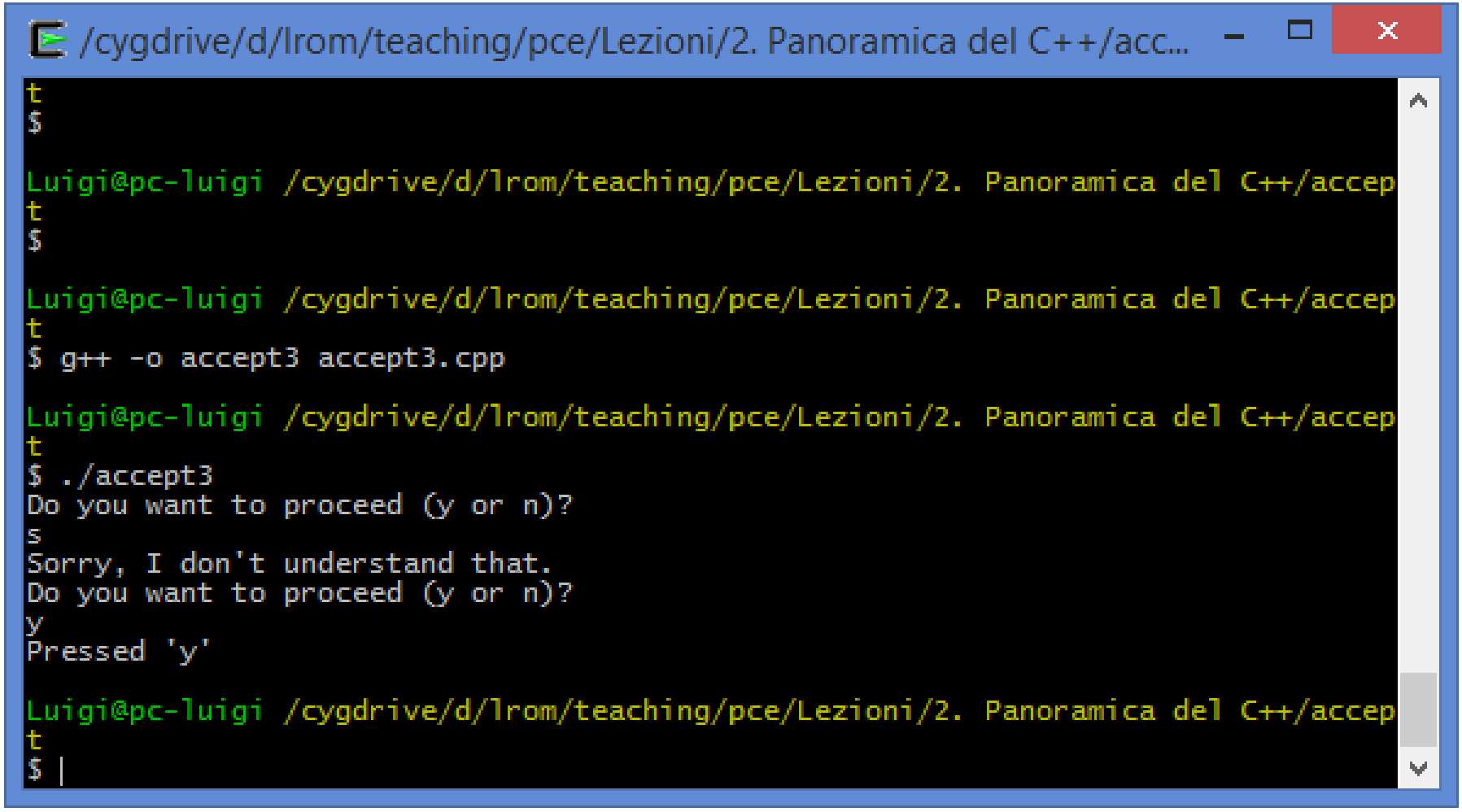
ACCEPT2 (): COMPILAZIONE ED ESECUZIONE



while: esegue istruzioni finché la condizione è vera

```
bool accept3()
{
        int tries = 1;
        while(tries < 4) {
                cout<<"Do you want to proceed (y or n)?\n";      // write question
                char answer = 0;
                cin>>answer;                                     // read answer
                switch(answer) {
                        case 'y':
                                return true;
                        case 'n':
                                return false;
                        default:
                                cout<<"Sorry, I don't understand that.\n";
                        tries=tries+1;
                }
        }
        cout<<"I'll take that for a no.\n";
        return false;
}
```

ACCEPT3 (): COMPILAZIONE ED ESECUZIONE

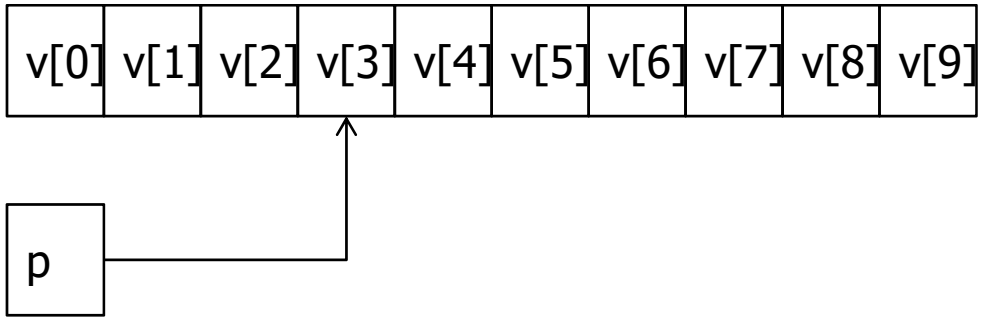


Un *array* è una struttura dati composta da più elementi dello stesso tipo

Una variabile *puntatore* memorizza l'indirizzo di un oggetto di un appropriato tipo

char v[10]; // array di 10 caratteri

char p; // puntatore a carattere*
p = &v[3];



- Si basa sulla suddivisione di procedure logicamente correlate e dei dati che manipolano in *moduli*
  - Ogni modulo può invocare un altro modulo
- Paradigma noto come principio del *data-hiding*

```
namespace Stack {    // interface
        void push(char);
        char pop();
}

void f()
{
        Stack::push('c');
        if(Stack::pop()!='c') error("impossible");
}
```

```
namespace Stack {    // implementation
        const int max_size = 200;
        char v[max_size];
        int top = 0;
        void push(char c) {
                /* check for overflow
                 and push c */
        }
        char pop() {
                /* check for underflow
                 and pop */
        }
}
```

Può essere usata per organizzare un programma in un insieme di frammenti semi-indipendenti

Header file *stack.h*

```
namespace Stack {    // interface
        void push(char);
        char pop();
}
```

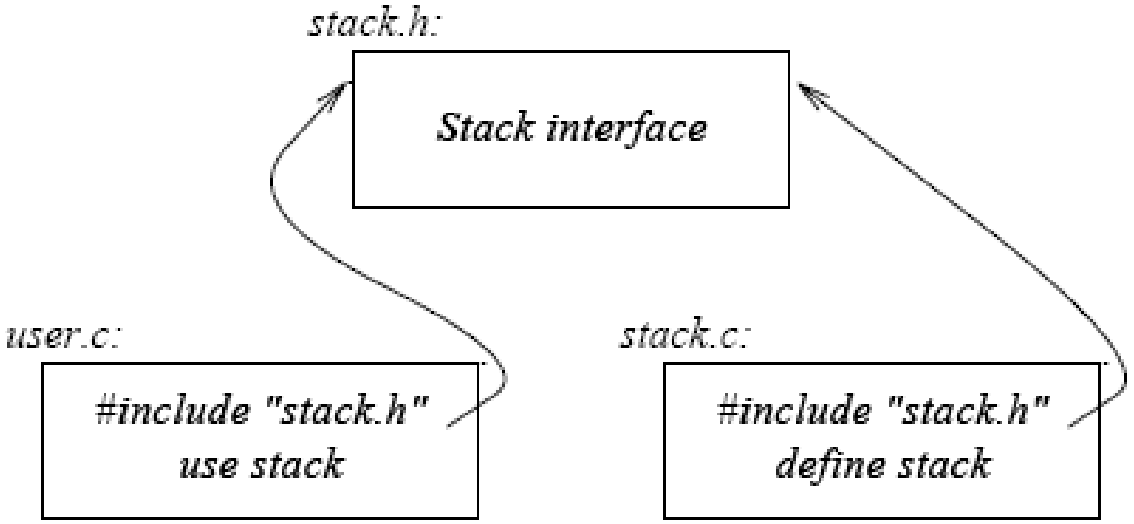
File *user.c*

```
#include "stack.h"
void f()
{
        Stack::push('c');
        if(Stack::pop()!='c') error("impossible");
}
```

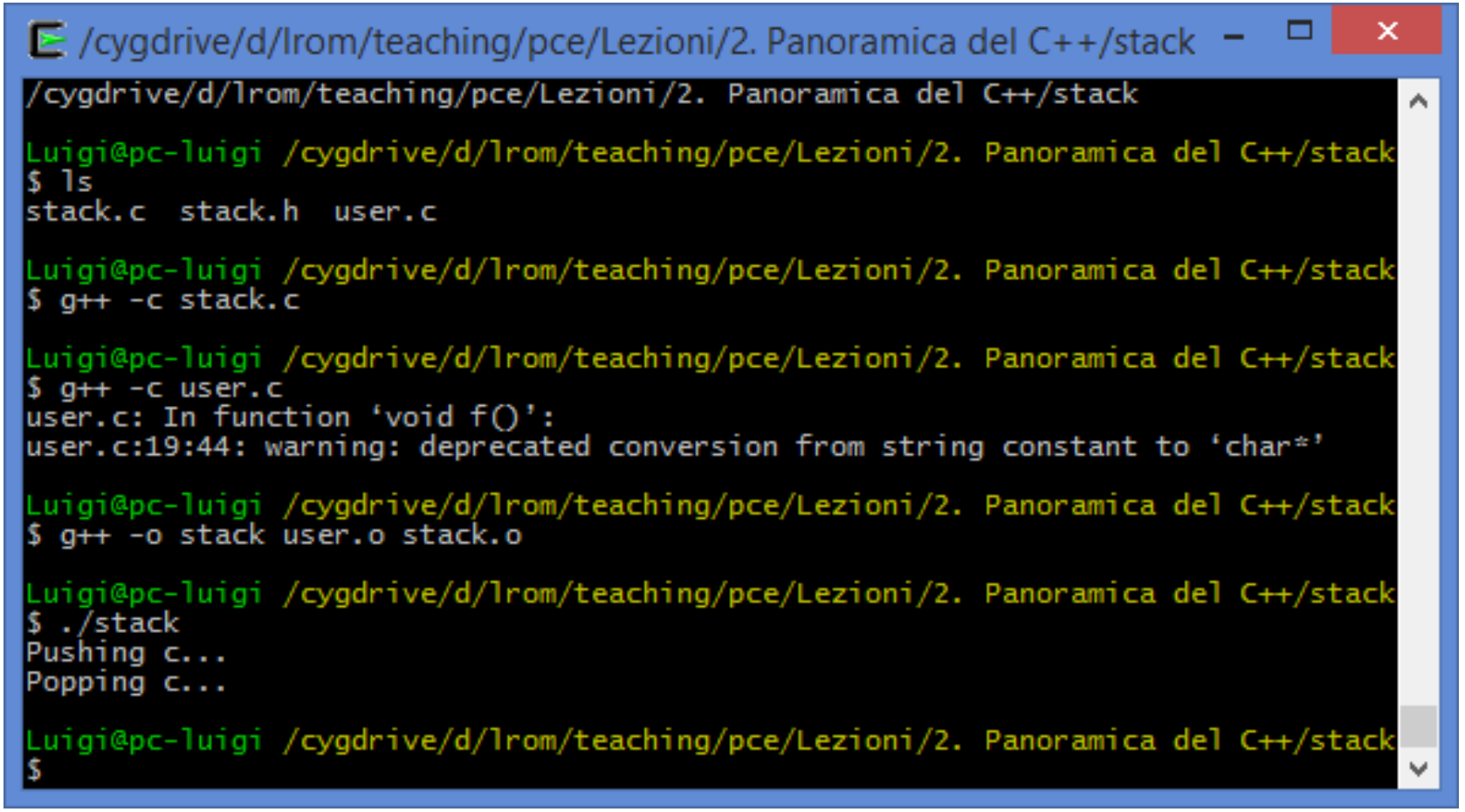
File *stack.c*

```
#include "stack.h"
namespace Stack {    // implementation
        const int max_size = 200;
        char v[max_size];
        int top = 0;
        void push(char c) {
                /* check for overflow
                 and push c */
        }
        char pop() {
                /* check for underflow
                 and pop */
        }
}
```

user.c e *stack.c* condividono le informazioni presenti in *stack.h*, ma sono indipendenti e possono essere compilati separatamente



STACK.C: COMPILAZIONE ED ESECUZIONE



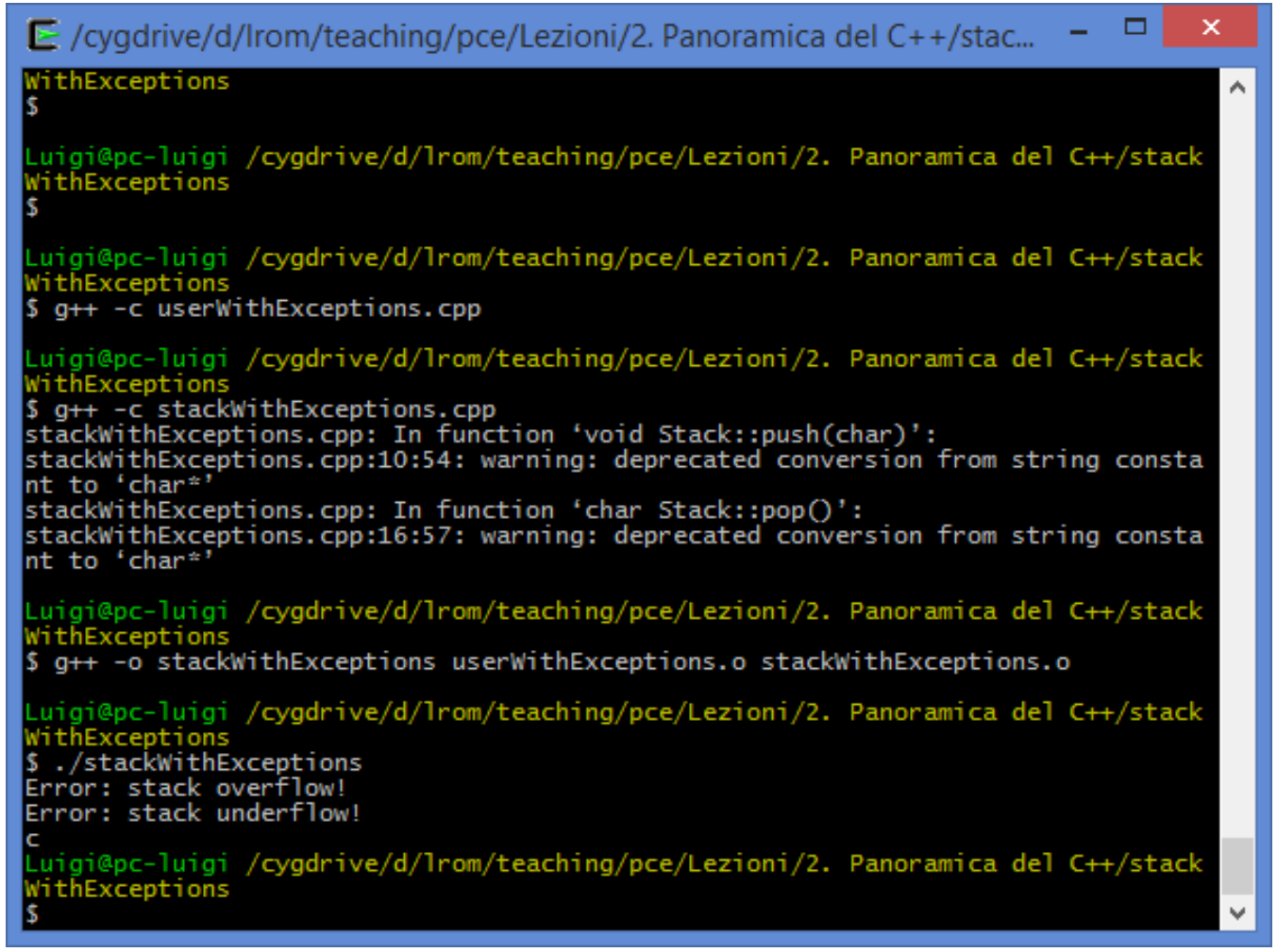
- I moduli devono rilevare eventuali *eccezioni* e comunicarle al modulo chiamante
- Il modulo chiamante può in tal modo mettere in atto azioni opportune

```
namespace Stack {    // interface
        void push(char);
        char pop();

        class Overflow { }; // type representing overflow exceptions
        class Underflow { }; // type representing underflow exceptions
}
```

```
void f() {
        // ...
        /* exceptions here are handled by the
        handler defined below */
        try {
                while(true) Stack::push('c');
        }
        catch(Stack::Overflow) {
                /* oops: stack overflow;
                take appropriate action */
        }
        // ...
        try {
                while(true) Stack::pop();
        }
        catch(Stack::Underflow) {
                /* oops: stack underflow;
                take appropriate action */
        }
        // ...
}
```

```
#include "stack.h"
namespace Stack {    // implementation
        const int max_size = 200;
        char v[max_size];
        int top = 0;
        void push(char c) {
                if(top == max_size)
                        throw Overflow();
                || push c
        }
        char pop() {
                if(top == 0)
                        throw Underflow();
                || pop c
        }
}
```

COMPILAZIONE ED ESECUZIONE



ASTRAZIONE DATI

- La programmazione modulare da sola non basta
- Un modulo utente non dovrebbe conoscere l'effettiva implementazione del modulo invocato
- È sufficiente che il modulo utente conosca solo l'interfaccia del modulo da invocare

ASTRAZIONE DATI: DEFINIZIONE DI TIPI NEI MODULI 1/3

- La programmazione modulare conduce alla centralizzazione di tutti i dati di un determinato tipo
- Un modulo deve occuparsi della gestione di tali dati

```
namespace Stack {
        struct Rep;          // definition of stack layout is elsewhere
        typedef Rep& stack;
        stack create();              // make a new stack
        void destroy(stack s);       // delete s
        void push(stack s, char c);  // push c onto s
        char pop(stack s);           // pop s
}
```

ASTRAZIONE DATI: DEFINIZIONE DI TIPI NEI MODULI 2/3

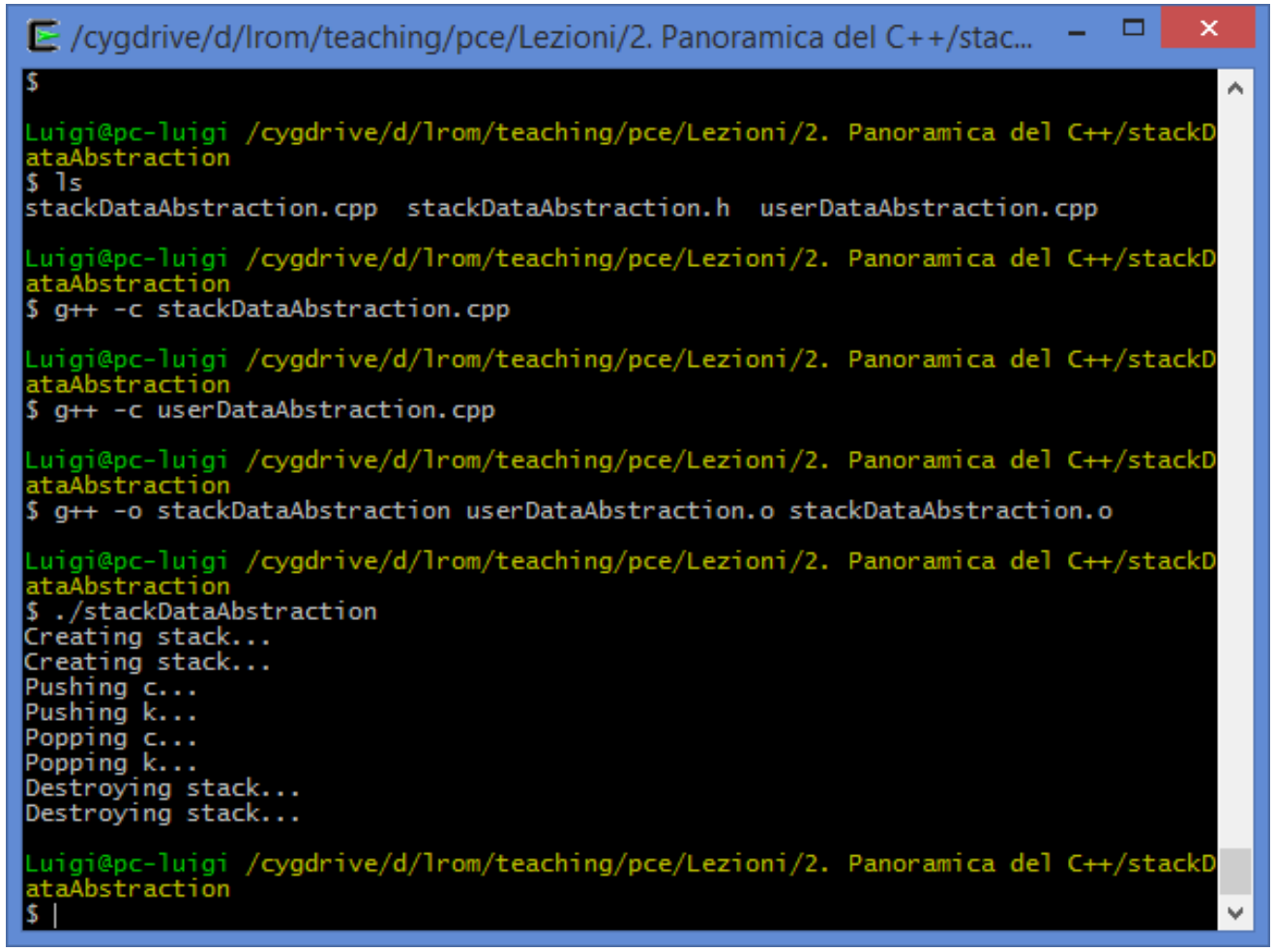
- Uno stack è identificato da *Stack::stack*
- In tal modo tutti i dettagli sono nascosti agli utenti

```
struct Bad_pop { };

void f() {
        Stack::stack s1 = Stack::create();      // make a new stack
        Stack::stack s2 = Stack::create();      // make another new stack
        Stack::push(s1,'c');
        Stack::push(s2,'k');
        if(Stack::pop(s1) != 'c') throw Bad_pop();
        if(Stack::pop(s2) != 'k') throw Bad_pop();
        Stack::destroy(s1);
        Stack::destroy(s2);
}
```

- Il tipo *Stack* può essere implementato in modi differenti

```
namespace Stack { // representation
        const int max_size = 200;
        struct Rep {
                char v[max_size];
                int top;
        };
        const int max = 16;   // maximum number of stacks
        Rep stacks[max];      // preallocated stack representations
        bool used[max];       // used[i] is true if stacks[i] is in use
}

void Stack::push(stack s, char c) { /* check s for overflow and push c */ }
char Stack::pop(stack s) { /* check s for underflow and pop */ }
Stack::stack Stack::create() {
        // pick an unused Rep, mark it used, initialize it, and return a reference to it
}
void Stack::destroy(stack s) { /* mark s unused */ }
```

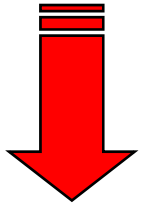
COMPILAZIONE ED ESECUZIONE



ASTRAZIONE DATI: TIPI DEFINITI DAGLI UTENTI 1/3

- Spesso la presentazione agli utenti dei tipi definiti nei moduli può variare in base alla rappresentazione del tipo

- Ma gli utenti devono essere isolati dalla conoscenza della rappresentazione del tipo



Definizione di *tipi di dati astratti*
(o *tipi definiti dagli utenti*)

ASTRAZIONE DATI: TIPI DEFINITI DAGLI UTENTI 2/3

- Lo stile di programmazione modulare non è sufficiente laddove vi sia la necessità di definire più oggetti di uno stesso tipo
- Una *classe* è un tipo di dato astratto che specifica la rappresentazione di un concetto ed una serie di operazioni su di esso

```
class complex {
        double re, im;
public:
        complex(double r, double i) { re = r; im = i; }    // construct complex from two scalars
        complex(double r) { re = r; im = 0; }              // construct complex from one scalar
        complex() { re = im = 0; }                                  // default complex: (0,0)
        friend complex operator+(complex a1, complex a2) {
                return complex(a1.re + a2.re, a1.im + a2.im);
        }
        friend complex operator-(complex, complex);        // binary
        friend complex operator-(complex);                 // unary
        friend complex operator*(complex, complex);
        friend complex operator/(complex, complex);
        friend bool operator==(complex, complex);          // equal
        friend bool operator!=(complex, complex);          // not equal
        // ...
};
```

La rappresentazione è privata

Costruttori

Accesso alle variabili private

Il compilatore converte gli operatori sugli oggetti *complex* in appropriate chiamate a funzioni

```
void f(complex z) {
        complex a = 2.3;
        complex b = 1/a;
        complex c = a + b *
        complex(1,2.3);
        || ...
        if (c != b) c = - (b/a)+2*b;
}
```

COMPILAZIONE ED ESECUZIONE


```
/cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex
$ pwd
/cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex
$ ls
complex.h  userComplex.cpp

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex
$ g++ -o userComplex userComplex.cpp

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex
$ ./userComplex
Re: 2
Im: 3.1

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/complex
$
```

- Il *costruttore* inizializza un oggetto appena creato
- Il *distruttore* elimina dalla memoria le variabili allocate con l'operatore *new*
- Il costruttore ed il distruttore sono implicitamente invocati

Interfaccia della classe Stack

```
class Stack {
                char* v;
                int top;
                int max_size;
public:
                class Underflow { };   // used as exception
                class Overflow { };    // used as exception
                class Bad_size { };    // used as exception
                Stack(int s);           // constructor
                ~Stack();               // destructor
                void push(char c);
                char pop();
};
```

ASTRAZIONE DATI: TIPI CONCRETI 2/2

Implementazione della classe Stack

```
Stack::Stack(int s) { // constructor
        top = 0;
        if(10000 < s) throw Bad_size();
        max_size = s;
        v = new char[s];  /* allocate elements on the
                             free store (heap, dynamic store)*/
}

Stack::~Stack() { // destructor
        delete[] v;    /* free the elements for
                          possible reuse of their space */
}

void Stack::push(char c) {
        if(top == max_size) throw Overflow();
        v[top] = c;
        top = top + 1;
}

char Stack::pop() {
        if(top == 0) throw Underflow();
        top = top - 1;
        return v[top];
}
```

Utilizzo della classe Stack

```
Stack s_var1(10);       // global stack with 10 elements
void f (Stack& s_ref, int i) // reference to Stack
{
        Stack s_var2(i); // local stack with i elements
        Stack* s_ptr = new Stack(20);   /* pointer to Stack
                                allocated on free store */
        s_var1.push('a');
        s_var2.push('b');
        s_ref.push('c');
        s_ ptr->push('d');
        // ...
}
```

- Se la rappresentazione cambia in maniera significativa, l'utente è costretto a ricompilare
- Per isolare completamente l'utente di una classe dai cambiamenti all'implementazione di quest'ultima, bisogna disaccoppiare l'interfaccia dalla rappresentazione ed evitare l'uso di variabili locali

```
class Stack {
public:
        class Underflow { };   // used as exception
        class Overflow { };    // used as exception
        virtual void push(char c) = 0;
        virtual char pop() = 0;
};
```

Classe astratta

Polimorfismo: le classi che derivano da *Stack* devono implementare queste funzioni

La funzione f() usa l'interfaccia Stack senza conoscere i dettagli implementativi

```
// Array_stack implements Stack
class Array_stack: public Stack {
        char* p;
        int max_size;
        int top;
public:
        Array_stack(int s);
        ~Array_stack();
        void push(char c);
        char pop();
};
```

```
void f(Stack& s_ref) {
        s_ref.push('c');
        if(s_ref.pop() != 'c')
                throw bad_stack();
}

void g() {
        Array_stack as(200);
        f(as);
}
```

La funzione f() è a conoscenza della sola interfaccia Stack, quindi continuerà a funzionare anche con una diversa implementazione di Stack

```
// List_stack implements Stack
class List_stack: public Stack {
        list<char> lc; // (standard library) list of characters
public:
        List_stack() { }
        void push(char c) { lc.push_front(c); }
        char pop();
};

char List_stack::pop() {
        char x = lc.front(); // get first element
        lc.pop_front(); // remove first element
        return x;
}
```

```
void h() {
        List_stack ls;
        f(ls);
}
```

COMPILAZIONE ED ESECUZIONE


```
/cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/stackAbstractClass
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/stackAbstractClass
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/stackAbstractClass
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/stackAbstractClass
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/stackAbstractClass
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/stackAbstractClass
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/stackAbstractClass
$

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/stackAbstractClass
$ ls
arrayStack.h  listStack.h  stackAbstractClass.h  userStackAbstractClass.cpp

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/stackAbstractClass
$ g++ -o userStackAbstractClass userStackAbstractClass.cpp

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/stackAbstractClass
$ ./userStackAbstractClass
Pushing Array_stack...
Pushing List_stack...

Luigi@pc-luigi /cygdrive/d/lrom/teaching/pce/Lezioni/2. Panoramica del C++/stackAbstractClass
$
```

- **Non c'è modo di adattare un tipo definito in un modulo a nuovi scopi se non modificando la sua definizione**
- **Questa situazione rende il codice poco flessibile**

```
class Point { /* ... */ };
class Color { /* ... */ };

enum Kind {circle, triangle, square}; // enumeration

class Shape {
        Kind k; // type field
        Point center;
        Color col;
        // ...
public:
        void draw();
        void rotate(int);
        // ...
};
```

```
void Shape::draw() {
        switch(k) {
            case circle:
                || draw a circle
                 break;
            case triangle:
                || draw a triangle
                break;
            case square:
                || draw a square
                break;
        }
}
```

I linguaggi orientati agli oggetti consentono di distinguere tra le proprietà generali di ogni *forma* dalle proprietà di uno specifico tipo di *forma*

```
class Shape {
        Point center;
        Color col;
        // ...
public:
        Point where() { return center; }
        void move(Point to) { center = to; /* ... */
draw(); }
        virtual void draw() = 0;
        virtual void rotate(int angle) = 0;
        // ...
};
```

Polimorfismo: tali metodi devono essere definite solo per forme specifiche

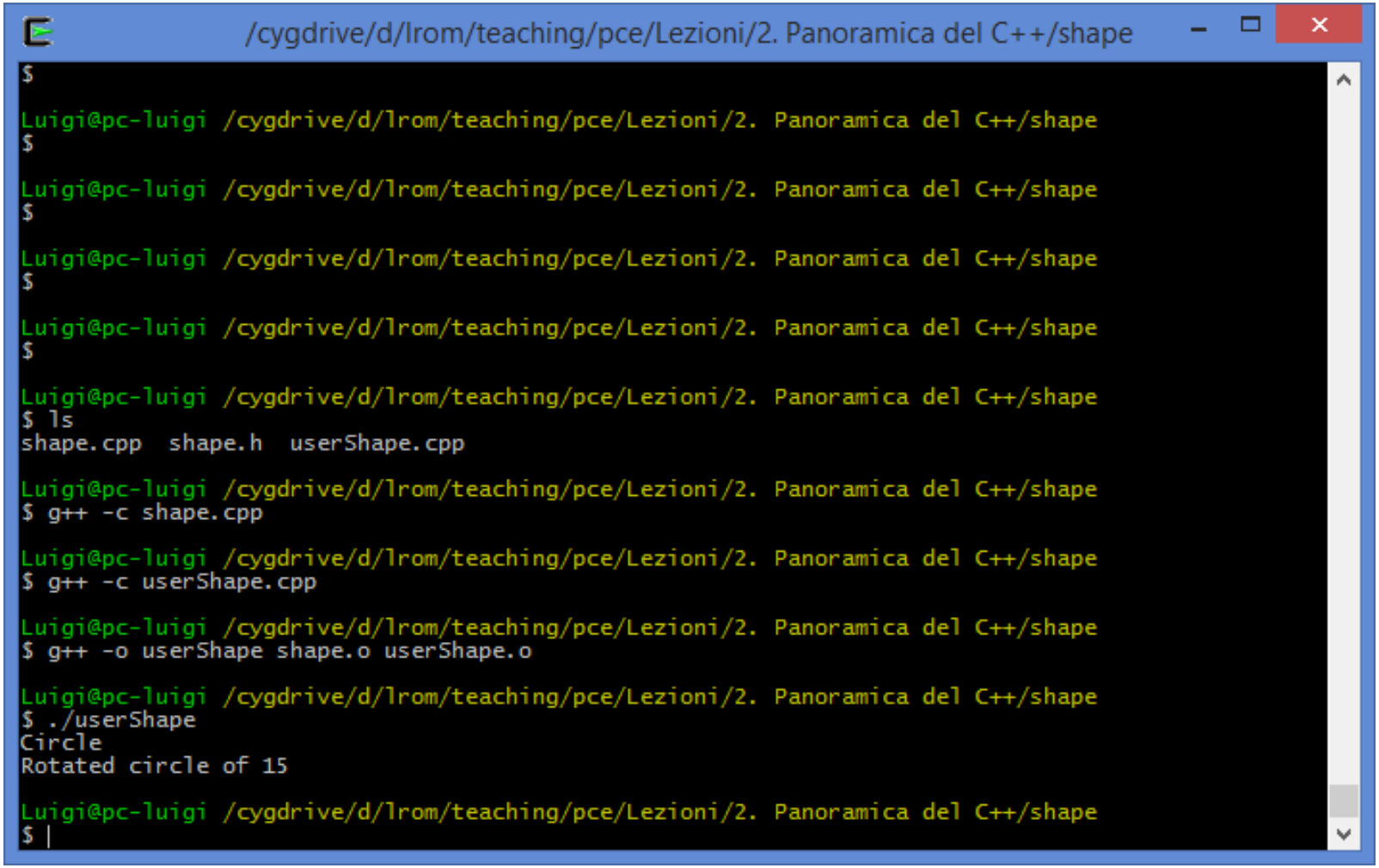
Possono essere definite funzioni generali che manipolano vettori di puntatori a *forme*

```
void rotate_all(vector<Shape*>& v, int angle) // rotate v's elements angle
degrees
{
        for(int i=0; i<v.size(); ++i ) v[i]->rotate(angle);
}
```

Per definire una *forma* particolare, è necessario prima indicare che essa è una *forma*

```
class Circle: public Shape {
        int radius;
public:
        void draw() { /* ... */ }
        void rotate(int) {} // yes, the null function
};
```

COMPILAZIONE ED ESECUZIONE



- È uno stile di programmazione che consente di *parametrizzare* classi o funzioni

- È un paradigma estremamente flessibile, dal momento che consente di definire ed implementare parti di codice indipendentemente dai dati trattati

- Una classe implementata per un particolare tipo può essere generalizzata dichiarandola *template* e rimpiazzando il tipo specifico con un parametro
- Questa classe è detta classe *contenitore*

```
template<class T> class Stack {
        T* v;
        int max_size;
        int top;
public:
        class Underflow { };
        class Overflow { };
        Stack (int s); // constructor
        ~Stack();  // destructor
        void push(T);
        T pop();
};
```

```
template<class T> void Stack<T>::push(T c) {
        if(top == max_size) throw Overflow();
        v[top] = c;
        top = top + 1;
}

template<class T> T Stack<T>::pop() {
        if(top == 0) throw Underflow();
        top = top - 1;
        return v[top];
}
```

Quando si istanzia un contenitore, è necessario specificarne il tipo

```
Stack<char> sc;            // stack of characters
Stack<complex> scplx;   // stack of complex numbers
Stack< list<int> > sli;     // stack of list of integers

void f() {
        sc.push('c');
        if(sc.pop() != 'c') throw Bad_pop();
        scplx.push(complex(1,2));
        if(scplx.pop() != complex(1,2)) throw Bad_pop();
}
```

- Il paradigma della programmazione generica può essere utilizzato anche per *parametrizzare* algoritmi

- Un approccio si basa sulla nozione di *sequenza* e sulla loro manipolazione per mezzo di *iteratori*

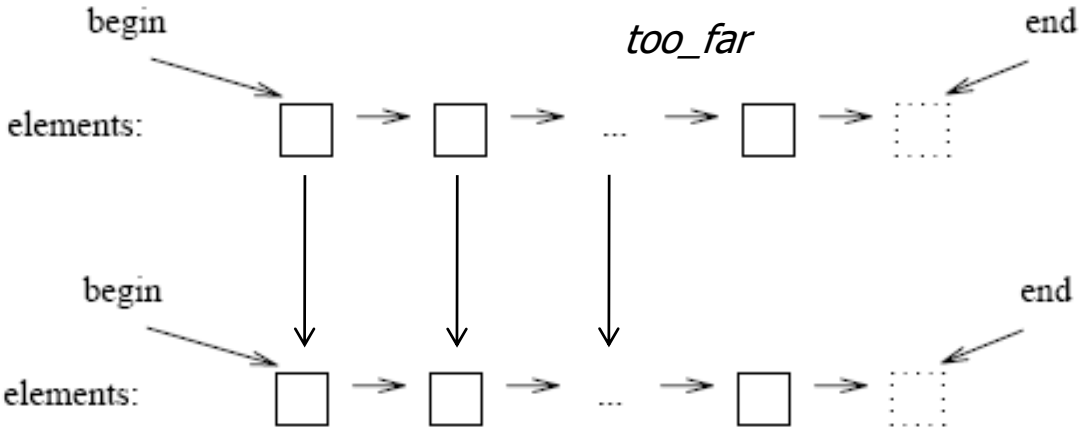
```
template<class In, class Out> void copy(In from, In too_far, Out
to) {
        while(from != too_far) {
                *to = *from;    // copy element pointed to
                ++to;           // next input
                ++from;         // next output
        }
}
```

* accesso ad un elemento attraverso un iteratore

++ incremento dell'iteratore in modo tale che punti al prossimo elemento

Sequenza *from*

Sequenza *to*



In questo modo è possibile copiare due contenitori dello stesso tipo o di tipo diverso

```
char vc1[200]; // array of 200 characters
char vc2[500]; // array of 500 characters

void f()
{
    copy(&vc1[0],&vc1[200],&vc2[0]);
}
```

```
complex ac[200];

void g(vector<complex>& vc, list<complex>&
lc)
{
    copy(&ac[0],&ac[200],lc.begin());
    copy(lc.begin(),lc.end(),vc.begin());
}
```

ALCUNI CONSIGLI…

- Niente panico! Tutto diventerà chiaro a tempo debito
- Non bisogna conoscere ogni dettaglio del C++ per scrivere buoni programmi
- Focalizzarsi sulle tecniche di programmazione, non sulle caratteristiche del linguaggio