TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science / Institute of Systems Architecture / Operating Systems

SYSTEMS PROGRAMMING
C++ INTRODUCTION

Alexander Warg

WHY C++ ?

- C++ is the language that allows to express ideas from the systems-programming area most direcly
- C++ is widely used in engineering areas
- C++ is available on almost any computer

WHAT (NOT) TO EXPECT

- Explanation of C++ constructs for real understanding: *C++ is not magic*
- Try to not to explain any obscure detail

NOT

- C++ standard library APIs
- ...

TECHNISCHE UNIVERSITÄT DRESDEN

OUTLINE

C / C++ TYPE SYSTEM

POINTERS AND REFERENCES

ARRAYS AND POINTER ARITHMETIC

BUILDING A LINKED LIST

INHERITANCE AND TYPE CONVERSION

OBJECTS, TYPES, AND VALUES

int x = 10;

‹type› ‹name› ‹initializer value›

GENERALLY

The type of an object determines which operations are allowed and their semantics

- x + y (Complex vs int)
- f(x) etc.

TYPES

BUILT-IN TYPES

bool — boolean type (true, false)

char — character type ('a', '4'...)

short, int, long, (long long)
signed integer types (0, 1, 2, -5 ...)

unsigned char .. unsigned long long
unsigned interger types (0, 1, 2 ...)

float, double
floating-point numbers (1.2, 3.4, 1.2e3 ...)

USER-DEFINED TYPES

follow soon

OBJECTS

- Some memory that can hold a value of a given type
- A variable is a named object
- A declaration names an object

```
int a = 7;
char c = 'x';
std::complex<double> z(1.0,2.0);
```

COMPOUND TYPES

ARRAYS int x[10]

FUNCTIONS void func(int p1, double z)

POINTERS int *ptr

REFERENCES int &ref = x; // alias for x

CLASSES, STRUCTS

UNIONS

ENUMERATIONS

POINTERS TO NON-STATIC MEMBERS

USER-DEFINED TYPES

struct, class

- compound data type aggregating one or multiple instances of other data types
- struct essentially is the same as a class
- operations (methods, operators) for the type

enum

- enumeration type with user-define constant values

union

- can contain different types at different times

CLASS, STRUCT, BITFIELD ....

PRACTICAL EXERCISE

- members
- visibility
  - *public* visible for all (default for *struct*)
  - *private* only inside the class (default for *class*)
  - *protected* inside the class and derived classes

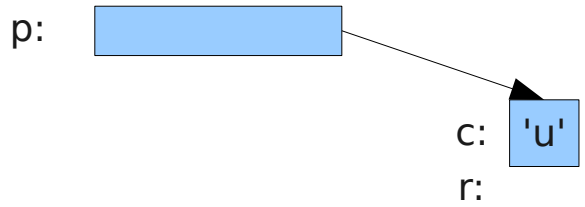
TYPEDEF vs USER-DEFINED TYPES

PRACTICAL EXERCISE

- class S { int val; }; vs typedef int S;
- typedef struct Thing { ... } Thing;
  - this is C not C++, however is allowed for compatibility
  - *struct Thing* already defines the type *Thing*

POINTERS / REFERENCES

```
char c = 'u';
char *p = &c;    // pointer to the object c
char &r = c;     // reference to object c (alias)
```

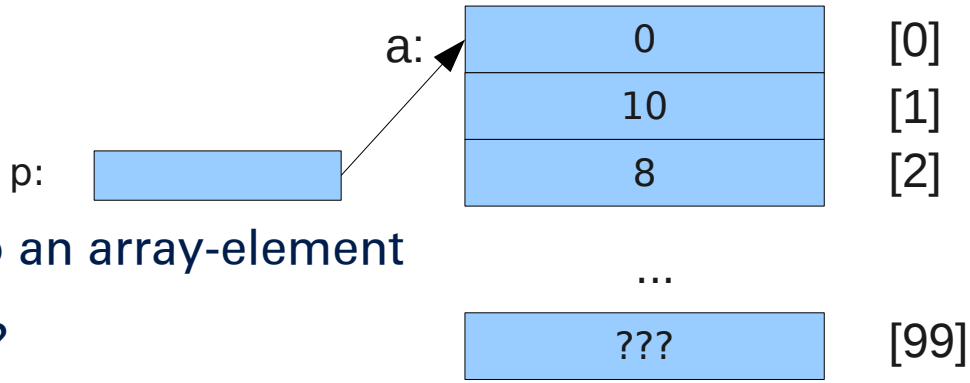


- pointers can be 'Null': p = 0
- references are always valid (technically comparable to a pointer)

ARRAYS & POINTER (ARITHMETIC)

```
int a[100] = { 0, 10, 8, ... };

int *p = a; // pointer can point to an array-element
p = p + 1; // what happens here ?
```




```
char c[100];
char *cp = c;
cp++;
```

TYPE QUALIFIERS

const int const x = 10;

- makes an object immutable

volatile int volatile x;

- defines x to have side effects or to be modified externally (asynchronously)

Examples...

STORAGE, SCOPE, LIFE CYCLE

```
int x;          // global variable, created at program start,
                // destroyed at program exit
void func()
{
 int x;         // local variable, created here,
                // destroyed on function return
}

static int x; // global storage, locally visible in this compilation unit

void func1()
{
 static int x; // global storage, visible in this function
}
```

... STORAGE, SCOPE, LIFE CYCLE

```
class Data
{
 int x;                    // object scope (life cycle of the respective
                           // Data object)
 int get_x() const         // object scope function (method)
 { return x; }

 static int z;             // class scope (global storage, global life cycle)
 static int get_z()        // class scope function
 { return z; }
};

// NOTE: define static class data at global scope
//                this must usually not be in a header file!
int Data::z;
```

HEAP STORAGE

```
// allocate object on dynamic heap
Data *d = new Data;

// destroy object explicitly (no garbage collection!)
delete d;
```

usually new **must not return NULL, so no check needed**

FUNCTIONS

```
int func(long x);       // declaration

int func(long x)        // definition
{
 return 5 * x / 23;
}

int func(long x, long y) // different function (overloading)
{
 return x * y;
}

char func(long x, long y); // error ?
```

CALL BY VALUE / REFERENCE

```
void func1(int p)   // call by value (the default)
{ p = p + 1; }

struct Data { int x; int y; }

void func2(Data data)      // call by value
{ data.x = 10; }

void func3(Data &data)   // call by reference
{ data.x = 10; }

void func4(Data const &data)     // call by const reference
{ data.x = 20; }

void func5(Data *data) // call by reference/pointer
{ data->x = 23; }
```

ONLY USE NON-CONST REFERENCES WHEN REALLY NEEDED

SPECIAL FUNCTIONS

CONSTRUCTORS

- Initialize an object
- Same name as class, no return type

DESTRUCTORS

- Free resources of an object
- Name: ~<class name>(), no return type, no parameters

OPERATORS

- Most operators in C++ can be overloaded (+, - ...)
- Will be explained eventually

EXERCISE: LINKED LISTS

Implement a linked list of complex numbers with the following functions:

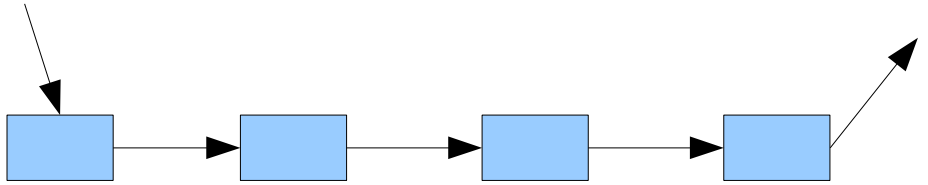
- insert given element at head
- insert given element at tail
- remove given element
- set / get complex value of given element
- search element whose value is the given complex number (if present)
- sum up all complex numbers in the list

(a specific form of the generic algebraic folding function over lists

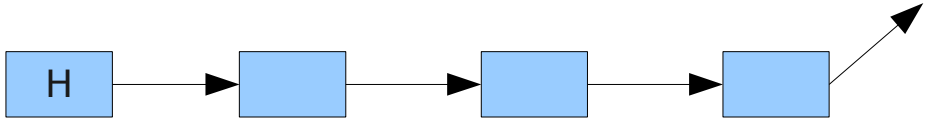
Which data structure is appropriate for a list with the above operations?

EXERCISE: LINKED LISTS

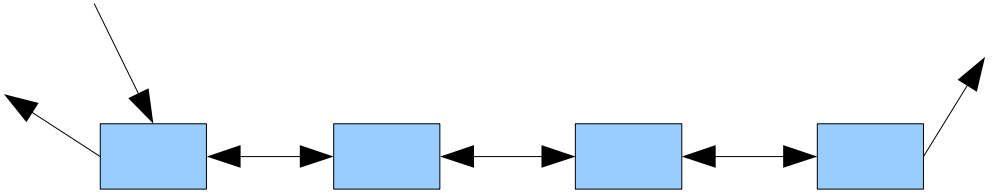
1) single-linked list



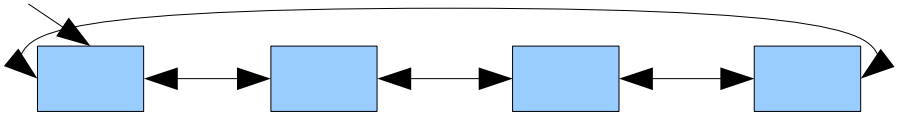
2) single-linked list with head element



3) double-linked list



4) double-linked list cyclic



5) double-linked list cyclic with head element