# Part 4: Code Modularization and Abstract Data Types

Contents [DOCUMENT FINALIZED]







C++ : feature available in C++ but not in C



## C Preprocessor

Week 6

- Compilation: transforming a textual program description into an executable form
- Preprocessor: separate first step in compilation:
  - Remove comments
  - Macro substitution (`#define`)
  - Conditional compilation (`#if`)
  - File inclusion (`#include`)
- Preprocessor directive: first non-white-space character in line is #
- Only one per line
- To view the preprocessor output which will be compiled, call `g++ -E file.c`



## Macro Substitution

```
#define N 1000
#define FOREVER for (;;)

FOREVER { if (i < N) i++; }
is translated into:
for (;;) { if (i < 1000) i++; }
```

- Syntax of a macro definition:
  `#define` <identifier> <replacement text>
- Subsequent occurrences of the identifier in places where identifiers are expected get replaced by the replacement text. E.g.

  `xxFOREVERxx` and `"FOREVER"` are not replaced!
  but
  `int x = N;` and `if (done) FOREVER`
  are
- Normally, the replacement text is the remainder of line. Lines can be continued by placing \ at the end.
- Scope is from point of definition to the end of current file.

## Macros With Parameters

```
#define is_even(x) ((x) % 2 == 0)

flag = is_even(y+2);
gets translated into
flag = ((y+2) % 2 == 0);

recursive = is_even(is_even(z));
gets translated into
recursive = ((((z) % 2 == 0)) % 2 == 0);
```

- Syntax:
  `#define` <ident>`(`<ident>`,`...`,`<ident>`)` <text>
- Macro parameters get replaced by actual arguments when macro is expanded
- Macro expansion is done recursively until no more matches are found



## More Macro Examples

```
#define FOR(i,n) for (i=0; i < (n); ++i)

FOR (i, 10) { foo(i); }
becomes
for (i=0; i < (10); ++i) { foo(i); }

#define MAX(a,b)  ((a)>(b)?(a):(b))
Not recommended! Multiple evaluations!
Also, use lots of () to ensure evaluation order!

MAX(a++,b++)
becomes
((a++)>(b++)?(a++):(b++))
OOPS! 2x a++,b++!
```

Big problem: the preprocessor doesn't know anything about the programming language we are using. It's a simple text replacement system.

In C++ there is hardly any reason for using parameterized macros anymore! Use constants and template/inline functions instead (later). These are type-safe - macros are not.



## Conditional Compilation

- Syntax & Semantics

```
#if <const-expr>  : true iff const-expr != 0
#ifdef <ident>    : true iff <ident> is def.
#ifndef <ident>   : true iff <ident> is undef.
#else             : alternative path
#elif <const-expr>: else-if condition
#endif            : end of #if statement
```

- <const-expr> can consist of macro names, integer constants, operators, parentheses, and `defined(<macro-name>)`
- `#error "text"`: generates error msg. `"text"`



## Application

```
#ifdef UNIX
  ... Unix code
#else
#ifdef WINDOWS
  ... Windows code
#else
  #error "Unsupported"
#endif
#endif
```

```
#define TEST 1

#if TEST
  ... test code
#endif
```

- Compiling parts of programs depending on constant expressions. If false, program text is skipped
- Useful for dealing with different environments and debugging
- Can pass macro definitions to gcc/g++ via -D option ("define"). E.g.

  ```
  g++ -DUNIX foo.c   // macro UNIX defined
  g++ -DFOO=3 foo.c // macro FOO = 3
  ```

  This way, programs can easily be adjusted to different environments. E.g.

  ```
  g++ -DUNIX foo.c     // UNIX version
  g++ -DWINDOWS foo.c // WINDOWS version
  ```

## File Inclusion

- Two forms:
  `#include "filename"`
  `#include <filename>`
- Line is replaced by the content of the file filename, which itself may contain `#include` lines
- `"filename"` : search for file begins in directory where the source program is located. If not found, search in system header directories
- `<filename>` : search file in system header directories
- Main purpose: including type information such as function and class declarations, and constants such as `M_PI`.



## #include Examples (1)

```
#include <stdio.h>
// the compiler now knows about function
// printf: function name, return type,
// and parameter types

#include "mystuff.h"
// your functions and data structures
// are now known to the compiler

// declared in /usr/include/stdio.h
printf("hello world\n");

// declared in ./mystuff.h
do_my_stuff();
```



## #include Examples (2)

C/C++ compilers reject double declarations such as this one:

```
struct Point { int x, y; };
struct Point { int x, y; };
```

When including files, this can easily happen. So the question becomes, how to avoid double inclusion?

mystuff.h:

```
#pragma once

#define FOR(i,n) for (i=0; i < (n); ++i)

int square(int x);
int swap(int &x, int &y);
int bitcount(unsigned int x);
```

The `#pragma once` preprocessor command instructs the preprocessor to only include the file once. I.e., if we attempt to include file `mystuff.h` twice, the second include directive will be ignored.



## Modular Programming

Modularization makes large programming projects manageable

Modular programs can be tested by FIRST validating individual modules and THEN testing how they work together. This GREATLY simplifies the testing task.

Moreover, when implemented properly, program parts can be compiled separately.

⇝ faster edit-compile-run cycle, because unchanged files don't have to be compiled again.

C/C++ way:

- Put constants, macros, function and type declarations in header (.h) files.
- Put function definitions in program modules (.c files) which can be compiled separately.
- Modules that make use of functions, types, macros, and constants need to `#include` the header files that contain their declaration.

## Separate Compilation of Modules

- For each module `file.c` call

  `g++ -c -o file.o file.c`

  This will compile (-c) `file.c` but not create an executable file. Instead it creates an object (.o) file `file.o` which contains executable code plus house-keeping data such as function names. Object files are not executable!

- Finally, link all project object files with

  `g++ -o proj file_1.o ... file_n.o`

  This will combine all object files and create executable file `proj`.

- Exactly one module must contain the `main` function.



Example

foo.h:

```
#pragma once

// comment: what does this
// file contain?

// the maximum foo value
const int FOO_MAX = 100;

// returns foo value
// corresponding to x
int foo(int x);
```

bar.h:

```
#pragma once

// comment ...
int bar(int x);
```

foo.c:

```
#include "foo.h"
#include "bar.h"

int foo(int x) {
  return bar(x);
}
```

bar.c:

```
#include "bar.h"

int bar(int x) {
  return 0;
}
```

foobar.c

```
#include "foo.h"
#include "bar.h"

int main()
{
  int x = foo(0);
  int y = bar(FOO_MAX);
  return 0;
}
```



## Compiling Modular Projects

The only files that need to be compiled are modules (.c files)

In the previous example we saw three .c files:

`foo.c   bar.c   foobar.c`

From those, executable `foobar` can be created in one step:

`g++ -o foobar foo.c bar.c foobar.c`

Here, the compiler will compile all three module files, which is wasteful if only one module file was changed.

Alternatively, an executable file can be created by compiling each module separately and linking object files:

```
g++ -c -o foo.o foo.c  # create .o files
g++ -c -o bar.o bar.c
g++ -c -o foobar.o foobar.c
g++ -o foobar foo.o bar.o foobar.o #link
```



## Compiling Modular Projects (Continued)

Now if say `foo.c` is changed, we only have to repeat the following steps:

```
g++ -c -o foo.o foo.c   # create .o file
g++ -o foobar foo.o bar.o foobar.o #link
```

The other object files don't have to be updated.

This can save a lot of time when working on big projects.

In addition, the process can be easily parallelized, i.e. multiple instances of g++ can be run simultaneously to compile your entire project much faster on multi-core architectures.

Now, if we only had a tool that detects whether a file has been updated and runs the compiler only on those.

Your wish just came true — meet makefiles ...

## Makefiles

- Purpose: executing shell commands according to file dependencies and time stamps
- Handy for compilation
  - Only compile modules that depend on recent changes
  - Easy to change compiler options globally
  - Adjust to operating system environments using conditional statements
- Can also be used for other tasks including
  - Cleaning up directories
  - Create pdf-file from Latex source
  - Generating html-documentation (doxygen)



## Makefiles (Continued)

- Rules (= file dependencies) and commands for updating files are stored in file commonly named `makefile` or `Makefile`. E.g.

  `make clean` to remove all object files (see p. 24)
- Run with: `make` or `make <target>`
  Executes commands for building first target in makefile or specific target
- Compile up to $k$ modules in parallel: Option `-j` $k$



## makefile Example

```
# executable foobar depends on foobar.o, foo.o, and bar.o
# generate it with g++ if one of those files is newer
# than foobar. foobar is "made" when make is called
foobar : foobar.o foo.o bar.o
<tab>   g++ -o foobar foobar.o foo.o bar.o

# foo.o depends on files foo.c foo.h bar.h
# if one of them is newer than foo.o call g++ to update it
foo.o  :  foo.c foo.h bar.h
<tab>   g++ -c -o foo.o foo.c

bar.o  :  bar.c bar.h
<tab>   g++ -c -o bar.o bar.c

foobar.o  :  foobar.c foo.h bar.h
<tab>   g++ -c -o foobar.o foobar.c

<tab> = invisible tab character
"make" updates first target (foobar)
```



## Variables

- Variables contain strings
- They can be used in command lines like so:

```
After

CC := g++
CCOPTS := -Wall -Wextra -Wconversion -O3

$(CC) $(CCOPTS)
later expands to
g++ -Wall -Wextra -Wconversion -O3
```

- Useful for changing compiler options globally, such as generating an executable file with debug information, or an optimized version

## Recursively Expanded Variables

`=` sets the value of a variable that is expanded recursively

```
FOO = $(BAR)
BAR = $(MOO)
MOO = moo
```

Then `$(FOO)` is expanded to `moo`



## Singly Expanded Variables

`:=` sets the value of a variable that is expanded once

```
X := foo
Y := $(X) bar
X := later
```

Then `$(Y)` is expanded to `foo bar`

Singly expanded variables contain no variable references (but their values at the time of definition)

Advantages: simpler behaviour, faster, can create lists!

E.g. `CCFLAGS := $(CCFLAGS) -O`

appends `-O` to an existing flag list



## Pattern Rules

- Generalized file dependencies + command(s)
- Example:
  - `%.o : %.c`
    `$(CC) $(CCOPTS) -c -o $@ $<`

    means: file `%.o` depends on file `%.c` for all words % (% = wildcard, similar to shell's *)
  - command is executed whenever a `file.o` is needed and `file.c` is more recent than `file.o`
- Command line(s) must start with tab character!
- Special variables are replaced by actual values when rule is applied
  - `$@` : rule target
  - `$<` : first prerequisite
  - `$^` : all prerequisites



## Complete makefile with Pattern Rule

```
CC := g++
WARN := -Wall -Wextra -Wconversion

# debug settings, uncomment when debugging
# CCOPTS := $(WARN) -g

# optimization setting, uncomment when done with debugging
CCOPTS := $(WARN) -O3 -DNDEBUG

# link executable when an .o file is newer
foobar : foobar.o foo.o bar.o
	$(CC) -o $@ $^

# how to compile .c files
%.o : %.c
	$(CC) $(CCOPTS) -c -o $@ $<

# remove object files and executable
clean:
	rm -rf *.o foobar

# file dependencies: paste output of g++ -MM *.c here
# if one file on the rhs is newer, use rule to update lhs
foobar.o : foobar.c foo.h bar.h
foo.o : foo.c foo.h bar.h
bar.o : bar.c bar.h
```

Invoke with `make`

## Makefile Session

Running the previous makefile for the first time will create files

`foo.o bar.o foobar.o foobar`

What will happen if you now run

```
touch bar.h
make
```

Checking file dependencies, make will recognize that `bar.h` is younger than

`foobar.o foo.o bar.o`

These files therefore have to be updated. Using the pattern rule `make` will invoke g++ three times.

The update process isn't over yet, because now

`foobar.o foo.o bar.o`

are younger than `foobar`.

`make` detects that and updates `foobar` by running

```
g++ -Wall ... -o foobar foobar.o foo.o bar.o
```



## GNU Make

- GNU make is installed on the lab machines
- Part of the GNU ("GNU is Not Unix") software collection (gcc, g++, gdb, gawk, gprof, emacs, ...)
- Free software implementation of original make plus many additional features
- Very powerful, general purpose tool
- Reading tutorials and documentation is highly recommended if you want to create more sophisticated makefiles, that — for instance — recompute file dependencies automatically.

  `www.gnu.org/software/make/manual`

- Interesting advanced reading dealing with managing large programming projects: search for

  "Recursive make considered harmful"



## Global Variables

Week 7

- Defined outside any { } block
- C data types initialized with default value 0
- Scope is entire program unless the `static` modifier is used to indicate that the variable's scope is local to the current module
- Should be avoided because of potential name conflicts and accidents (every program part can change global variables!)
- Static and global variables are placed in the process data memory segment, which is separate from the stack and memory heap.



## Global Variable Example

```
int global; // initialized with 0 (*)

// everyone can change it!
float global_pi = 3.1415926535;

// const prevents this!
const float global_e = 2.718;

// initialized to 0
static int I_am_local_to_the_current_file;

int main()
{
  float global; // (**) (masks (*)) uninitialized

  global = 5;    // changes local variable (**)
  global_pi = 0.0; // possibly not intended
}
```

## Global Variables and Multiple Modules

main.c:

```
#include "global.h"

int main()
{
  global_val = 1.0;
  ...
}
```

global.h:

```
#pragma once

// declaration
extern int global_val;
```

global.c:

```
// definition

int global_val;
...
```

Global variables must be defined in one .c file

They must be declared in a header file as `extern`

Modules that use global variables must include the header file in which they are declared.



## Static Local Variables

```
int do_something()
{
  // assignment only done once
  static int number_of_calls = 0;

  ++number_of_calls;

  if ((number_of_calls % 100) == 0) {
    // print something every 100-th call ...
  }
}
```

- Static function variables are global variables in disguise
- `static` modifier
- Initialized when the function is called for the first time
- Static function variables outlive the duration of the function call and keep their values between calls!



## Abstract Data Types (1)

When reasoning about algorithms it is convenient to gloss over implementation details, as they are often immaterial.

For instance, consider a replacement for C-arrays, which can be unsafe and lose size information when passed to functions.

What operations does our replacement — call it Vector — need to support?

- init($n$) : initialize vector so that it holds $n$ elements
- set($i$, $v$) : set element $i$ to $v$
- get($i$) : get element $i$
- size() : return number of elements
- free() : return memory to operating system

This list formulates the <u>interface</u> between user programs and the <u>implementation</u> of these functions, which is not important to users of Vector, as long as the function's time and space requirements are suitable.



## Abstract Data Types (2)

In this example, for a C-array replacement we would require that all functions above run in constant time, i.e. the time spent in these functions is independent of the Vector size, and that the memory requirement is linear in the size of the Vector.

The big advantage of the interface-implementation separation is that users just need to know the function specifications, but not the implementation details.

This creates flexibility when implementing software libraries: if the interface specifications stay fixed, the implementation can be changed without interfering with user code. I.e., the programs users write don't have to be changed when the library implementer improves his code.

In C, interface declarations are stored in header (.h) files in form of function and type declarations. Users only have to look at those files to learn about available functions and data types.

## Abstract Data Types (3)

Implementations in form of function definitions are stored in module (.c) files. These are compiled and the generated machine code is often archived in libraries.

Users don't need to know implementation details, and libraries can be updated without users having to change their code.

In what follows we will see examples of abstract data types and how they can be implemented in C and C++.



## Vector Application (C)

`main.c`

```
#include "Vector.h"
#include <stdio.h>

int main()
{
  const int N = 10000;
  Vector v;      // not initialized!

  v_init(&v, N); // allocate N elements

  // set them
  for (int i=0; i < N; i++) {
    v_set(&v, i, 2*i);
  }
  // print them
  for (int i=0; i < N; i++) {
    printf("%d\n", v_get(&v, i));
  }
  v_free(&v);    // free elements
  return 0;
}
```

Compile with `gcc -std=c99 main.c Vector.c`



## Vector Declaration in C

`Vector.h`

```
#pragma once

// Implementation detail! In a perfect world this
// information would be hidden from Vector users
typedef struct {
  int *elems;
  int n;
} Vector;

// Interface declaration. This information is
// all what Vector users need to know
// allocates memory for n elements
void v_init(Vector *p, int N);
// frees memory
void v_free(Vector *p);
// returns number of elements
int v_size(const Vector *p);
// returns i-th element
int v_get(const Vector *p, int i);
// sets i-th element to v
void v_set(Vector *p, int i, int v);
```

This is what Vector users need to know. The actual implementation in `Vector.c` isn't important.



## Vector Implementation in C (1)

`Vector.c`

```
#include "Vector.h"
#include <stdlib.h>

// allocates memory for n elements
void v_init(Vector *p, int N)
{
  p->elems = malloc(N * sizeof(p->elems[0]));
  p->n = N;
}

// frees memory
void v_free(Vector *p)
{
  free(p->elems);
}

// returns number of elements
int v_size(const Vector *p)
{
  return p->n;
}
```

## Vector Implementation in C (2)

Vector.c

```
#include <assert.h>

// returns i-th element
int v_get(const Vector *p, int i)
{
  assert(i >= 0 && i < p->n);
  return p->elems[i];
}

// sets i-th element to v
void v_set(Vector *p, int i, int v)
{
  assert(i >= 0 && i < p->n);
  p->elems[i] = v;
}
```



## Issues

```
Vector v;
v_init(&v, 100);
...
int x = v_get(&v, i);
...
v_free(&v);
```

What if we forget to initialize or free Vectors?

It would improve software quality if those functions are called automatically.

Also, `v_get(&v, i)` is awkward — `v[i]` would be much better.

To address these issues C++ features classes, which can be viewed as generalized structs.



## C++ Classes

Structures are special cases of classes: every struct is a class

Classes provide additional functionality, some of which impose run-time overhead:

- Member functions
  e.g. `v.foo();` calls member function `foo` on `v`;
  C-equivalent: `foo(&v);`
- Automatic construction and destruction
- Access restrictions
  (hide data and member functions from users)
- Polymorphism
  (same function name, different function called)
- Inheritance
  (code reuse in derived classes)
- Operator overloading
  (e.g., define code for Vector [ ] operator)

In this course, we will only discuss member functions, constructors, and destructors.



## Vector Application (C++) (1)

main.c

```
#include "Vector.h"
#include <stdio.h>

int main()
{
  const int N = 10000;
  Vector v(N); // Vector created on stack,
               // also calls Vector constructor
  v.n = 5;     // illegal, n is private

  // set elements
  for (int i=0; i < N; i++) {
    v.set(i, 2*i); // sets element i to 2*i in v
  }
  // print them
  for (int i=0; i < N; i++) {
    printf("%d\n", v.get(i));
  }
  // Vector v goes out of scope => its destructor
  // is called, freeing memory automatically!
  return 0;
}
```

Compile with `g++ main.c Vector.c`

## Vector Application in C++ (2)

main.c

```
#include "Vector.h"

int main()
{
  // Allocates enough memory to hold Vector
  // object and then calls Vector constructor
  // which in turn allocates 10 integers.
  Vector *p = new Vector(10);

  // calls set on object *p; sets element 5 to 0
  p->set(5, 0);

  // First calls Vector destructor freeing
  // 10 integers and then releases memory
  // p points to (the Vector object itself)
  delete p;

  Vector *q = new Vector(20);

  // Memory leak: q goes out of scope and we
  // lose access to the Vector. Because q is
  // just a pointer, ~Vector is not called!
  return 0;
}
```



## Vector Declaration in C++

Vector.h

```
#pragma once

class Vector
{
private:
  // data members hidden from users
  int *elems;
  int n;

public:
  // user accessible member functions
  // constructor: allocate n elements
  // called when Vector is created
  Vector(int n);
  // destructor: called when Vector
  // is going out of scope or is destroyed
  ~Vector();
  // returns number of elements
  int size();
  // returns i-th element
  int get(int i);
  // sets i-th element to v
  void set(int i, int v);
};
```



## Vector C++ Implementation (1)

Vector.c

```
#include "Vector.h"

// Vector constructor: allocate elements
// Vector:: denotes the class context
// we implement functions for
Vector::Vector(int N)
{
  elems = new int[N];
  n = N;
}

// destructor: free memory
Vector::~Vector()
{
  delete [] elems;
}

// return number of elements
int Vector::size()
{
  return n;
}
```



## Vector C++ Implementation (2)

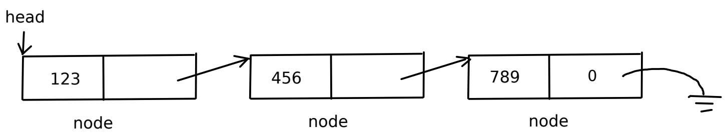
Vector.c

```
#include <assert.h>

// returns i-th element
int Vector::get(int i)
{
  assert(i >= 0 && i < n);
  return elems[i];
}

// sets i-th element to v
void Vector::set(int i, int v)
{
  assert(i >= 0 && i < n);
  elems[i] = v;
}
```

## Singly Linked Lists




We consider a singly linked list data structure implementing the following abstract functions:

- init() : empties list
- free() : frees memory and empties list
- add_head(data) : adds node at front with data
- get_head() : returns pointer to head (0 if empty)
- remove_head() : removes head node
- size() : returns number of list nodes

Nodes support the following functions

- set(data) : sets node data
- get() : gets node data
- next() : returns pointer to next node, or 0 if end of list is reached



## SList Application in C

main.c

```
#include "SList.h"
#include <stdio.h>

int main()
{
  SList list;     // components undefined!
  sl_init(&list); // list now empty

  // populate list
  for (int i=0; i < 10; i++) {
    sl_add_head(&list, i);
  }

  // print list; what is the output?
  Node *p = sl_get_head(&list);
  while (p) {
    printf("%d\n", n_get(p));
    p = n_next(p);
  }

  sl_free(&list); // free nodes, list now empty
}
```



## SList Declaration in C (1)

SList.h

```
#pragma once

// Node data structure
struct Node {
  int data;           // data associated with node
  struct Node *succ;  // pointer to successor node
};

typedef struct Node Node;

// Node functions
void n_set(Node *p, int data);
int  n_get(const Node *p);
Node *n_next(Node *p);

// SList data structure
typedef struct {
  Node *head;         // pointer to head node
  int n;              // number of nodes
} SList;
```



## SList Declaration in C (2)

```
// Initialize list: no nodes
void sl_init(SList *p);

// frees node memory
void sl_free(SList *p);

// add node containing data at front
void sl_add_head(SList *p, int data);

// returns pointer to head node
Node *sl_get_head(const SList *p);

// removes head node
void sl_remove_head(SList *p);

// return number of nodes
int sl_size(const SList *p);
```

## SList Implementation in C (1)

SList.c

```
#include "SList.h"
#include <stdlib.h>
#include <stdio.h>

// set node data
void n_set(Node *p, int data)
{
  p->data = data;
}

// return node data
int n_get(const Node *p)
{
  return p->data;
}

// return pointer to next node
Node *n_next(const Node *p)
{
  return p->succ;
}
```



## SList Implementation in C (2)

```
// initialize list: no nodes
void sl_init(SList *p)
{
  p->head = 0;
  p->n    = 0;
}

// frees all node memory and empties list
void sl_free(SList *p)
{
  while (p->head) {
    sl_remove_head(p);
  }
}

// return number of nodes
int sl_size(const SList *p)
{
  return p->n;
}
```



## SList Implementation in C (3)

```
// add node containing data at front
void sl_add_head(SList *p, int data)
{
  Node *q = malloc(sizeof(Node)); // new node
  q->data = data;          // store data in new node
  q->succ = p->head;       // old head is q successor
  p->head = q;             // new node is head
  p->n++;                  // one more node
}

// returns pointer to head node
Node *sl_get_head(const SList *p)
{
  return p->head;
}

// removes head node
void sl_remove_head(SList *p)
{
  if (!p->head) {          // no node?
    printf("can't remove head from empty list!\n");
    exit(1);               // stop program
  }
  p->n--;                  // one less node
  Node *q = p->head->succ; // save pointer to successor
  free(p->head);           // free head node
  p->head = q;             // head successor now head
}
```



## SList Application in C++

main.c

```
#include "SList.h"
#include <stdio.h>

int main()
{
  SList list;     // initialized!

  // populate list
  for (int i=0; i < 10; i++) {
    list.add_head(i);
  }

  // print list; what is the output?
  Node *p = list.get_head();
  while (p) {
    printf("%d\n", p->get());
    p = p->next();
  }

  // list nodes automatically freed
  return 0;
}
```

## SList Declaration in C++ (1)

SList.h

```
#pragma once

// Node data structure
// struct = class with public members
struct Node
{
  // data
  int data;      // data associated with node
  Node *succ;    // pointer to successor node

  // member functions
  void set(int data);
  int  get() const;
  Node *next();
};
```



## SList Declaration in C++ (2)

```
// SList data structure
class SList
{
private:
  Node *head; // pointer to head node
  int n;      // number of nodes

public:
  // constructor: no nodes
  SList();
  // destructor
  ~SList();

  // free all nodes and empty list
  void free();

  // add node containing data at front
  void add_head(int data);

  // returns pointer to head node
  Node *get_head();

  // removes head node
  void remove_head();

  // return number of nodes
  int size();
};
```



## SList Implementation in C++ (1)

SList.c

```
#include "SList.h"
#include <stdlib.h>
#include <stdio.h>

// set node data
void Node::set(int x)
{
  data = x;
}

// return node data
int Node::get() const
{
  return data;
}

// return pointer to next node
Node *Node::next()
{
  return succ;
}
```



## SList Implementation in C++ (2)

```
// constructor: initialize list, no nodes
SList::SList()
{
  head = 0;
  n = 0;
}

// destructor: free all nodes
SList::~SList()
{
  free();
}

// frees all node memory and empties list
void SList::free()
{
  while (head) {
    remove_head();
  }
}

// return number of nodes
int SList::size()
{
  return n;
}
```

## SList Implementation in C++ (3)

```
// add node containing data at front
void SList::add_head(int data)
{
  Node *q = new Node;   // new node
  q->data = data;       // store data in new node
  q->succ = head;       // old head is q successor
  head = q;             // new node is head
  n++;                  // one more node
}

// returns pointer to head node
Node *SList::get_head()
{
  return head;
}

// removes head node
void SList::remove_head()
{
  if (!head) {          // no node?
    printf("can't remove head from empty list!\n");
    exit(1);            // stop program
  }
  n--;                  // one less node
  Node *q = head->succ; // save pointer to successor
  delete head;          // free head node
  head = q;             // head successor now head
}
```



## Summary

C and C++ allow us to create abstract data types that separate interfaces from implementations.

With this, implementations can change without influencing user code.

Users just need to consult header (.h) files to learn how to use functions and data types. If interested, they can also look at the implementation (.c files), but that's not required.

C solutions are more clumsy and error prone because structs need to be initialized and freed manually.

C++'s classes support constructors and destructors, which are functions that are invoked automatically when class objects are created or destroyed, respectively.