Modularity and Data Abstraction

Procedural Abstraction

- When programs get large, certain **disciplines of structuring** need to be followed rigorously. Otherwise, the programs become complex, confusing and hard to debug.

- In your first programming course you learned the benefits of **procedural abstraction (διαδικαστική αφαίρεση).** When we organize a sequence of instructions into a function $F(x_1, \ldots, x_n)$, we have a named unit of action.

- When we later on use this function F, we only need to know **what** the function does, not **how** it does it.

Procedural Abstraction (cont'd)

- **Separating the what from the how** is an act of **abstraction (αφαίρεση)**. It provides two benefits:
  - Ease of use
  - Ease of modification

Information Hiding

- In your first programming course, you have also learned the benefits of having **locally defined variables**.
- This is an instance of **information hiding (απόκρυψη πληροφορίας)**.
- It has the advantage that local variables do not interfere with identically named variables outside the function.
- Abstraction and information hiding in a programming language are greatly enhanced with the concept of **module (ενότητα).**

Modules and Abstract Datatypes

- A **module** is a unit of organization of a software system that packages together a collection of entities (such as **data** and **operations**) and that carefully controls what external users of the module can see and use.
- Modules have ways of hiding things inside their boundaries to prevent external users from accessing them. This is called **information hiding**.
- **Abstract data types (αφαιρετικοί τύποι δεδομένων, ADTs)** are collections of objects and operations that present well defined **interfaces (διεπαφές)** to their users, meanwhile hiding the way they are represented in terms of lower-level representations.
- Modules can be used to **implement abstract data types**.

Modules (cont'd)

- Many modern programming languages offer **modules** that have the following important features:
  - They provide a way of grouping together related data and operations.
  - They provide clean, well-defined interfaces to users of their services.
  - They hide internal details of operation to prevent interference.
  - They can be separately compiled.

Modules (cont'd)

- Modules are an important tool for **"dividing and conquering"** a large software task by combining separate components that interact cleanly.
- They ease **software maintenance (συντήρηση λογισμικού)** by allowing changes to be made locally.

Encapsulation

- When we have features like modules in programming languages, we use the term **encapsulation** (**ενθυλάκωση**, the hidden local entities are **encapsulated** and a module is a **capsule**).

Modules in C

- By means of careful use of **header files,** we can arrange for separately compiled C program files to have the above four properties of modules.

- In this way C modules are similar to **packages** or **modules** in other languages such as Modula-2 and Ada.

Modules in C (cont'd)

- A C module `M` consists of two files `MInterface.h` and `MImplementation.c` that are organized as follows.
- The file `Minterface.h`:

```
/*------<the text for the file MInterface.h starts here>---------- */

     (declarations of entities visible to external users of the module)

/*---------------<end of file MInterface.h>-------------------------*/
```

Modules in C (cont'd)

- The file `MImplementation.c`:

```
/*-------<the text for the file Mimplementation.c starts here>------*/

#include <stdio.h>
#include "MInterface.h"

     (declarations of entities private to the module plus the)
     (complete declarations of functions exposed by the module)

/*----------------<end of file MImplementation.c>--------------------*/
```

The Interface file

- `MInterface.h` is the **interface** file.
- It declares all the entities in the module that are **visible** to (and therefore usable by) the external users of the module.
- Such visible entities include **constants, typedefs, variables** and **functions**. Only the prototype of each visible function is given (and only the argument types, not the argument names).
- The book by Standish recommends that declarations of functions in the interface file are **"extern"** declarations. This is not necessary so we will not follow it.

The Implementation File

- `MImplementation.c` is the **implementation** file.
- It contains all the **private entities** in the module, that are not visible to the outside.
- It contains the **full declarations and implementations** of functions whose prototypes have been given in the interface file.
- It **includes** (via `#include`) the user interface file.

The Main Program

- A **main program (client program)** that uses two modules `A` and `B` is organized as follows:

```
#include <stdio.h>
#include "ModuleAInterface.h"
#include "ModuleBInterface.h"

(declarations of entities used by the main program)

int main(void)
{
   (statements to execute in the main program)
}
```

Separate Compilation

- We can compile the module and the client program **separately:**

```
gcc -c MImplementation.c -o M.o
gcc -c ClientProgram.c -o ClientProgram.o
gcc M.o ClientProgram.o –o ClientProgram.exe
```

With the first two commands, we compile the C files to produce **object files**. Then, the object files are **linked** to produce the final executable.

Priority Queues – An Abstract Data Type

- A **priority queue** is a container that holds some prioritized items. For example, a list of jobs with a deadline for processing each one of them.

- When we remove an item from a priority queue, we always get the item with highest priority.

Defining the ADT Priority Queue

- A **priority queue** is a finite collection of items for which the following operations are defined:
  - **Initialize** the priority queue, *PQ*, to the empty priority queue.
  - Determine whether or not the priority queue, *PQ*, is **empty**.
  - Determine whether or not the priority queue, *PQ*, is **full**.
  - **Insert** a new item, *X*, into the priority queue, *PQ*.
  - If *PQ* is non-empty, **remove** from *PQ* an item *X* of highest priority in *PQ*.

A Priority Queue Interface File

```
/* this is the file PQInterface.h      */

#include "PQTypes.h"
/* defines types PQItem and PriorityQueue */

void Initialize (PriorityQueue *);
int Empty (PriorityQueue *);
int Full (PriorityQueue *);
void Insert (PQItem, PriorityQueue *);
PQItem Remove (PriorityQueue *);
```

Sorting Using a Priority Queue

- Let us now define an array `A` to hold ten items of type `PQItem`, where `PQItem`s have been defined to be integer values, such that bigger integers have greater priority than smaller ones:

```
typedef int PQItem;
typedef PQItem SortingArray[10];
SortingArray A;
```

- We can now use a priority queue to sort the array `A`.
- We can successfully use the ADT priority queue whose interface was given earlier **without having to know any details of its implementation.**

Sorting Using a Priority Queue (cont'd)

```
/* this is the main program */

#include <stdio.h>
#include "PQInterface.h"

typedef PQItem SortingArray[MAXCOUNT];
/* Note: MAXCOUNT is 10 */

void PriorityQueueSort(SortingArray A)
{
   int i;
   PriorityQueue PQ;

   Initialize(&PQ);
   for (i=0; i<MAXCOUNT; ++i) Insert(A[i], &PQ);
   for (i=MAXCOUNT-1; i>=0; --i) A[i]=Remove(&PQ);
}
```

Sorting Using a Priority Queue (cont'd)

```
int SquareOf(int x)
{
   return x*x;
}

int main(void)
{
   int i; SortingArray A;

   for (i=0; i<10; ++i){
      A[i]=SquareOf(3*i-13);
      printf("%d ",A[i]);
    }
    printf("\n");

    PriorityQueueSort(A);

    for (i=0; i<10; ++i) {
      printf("%d ",A[i]);
    }
    printf("\n");

    return 0;
}
```

Implementations of Priority Queues

- We will present two implementations of a priority queue:
  - Using sorted linked lists
  - Using unsorted arrays

The Priority Queue Data Types

In the **sorted linked list case**, the file `PQTypes.h` can be defined as follows:

```
#define MAXCOUNT 10

typedef int PQItem;

typedef struct PQNodeTag {
           PQItem   NodeItem;
           struct PQNodeTag *Link;
        } PQListNode;

typedef struct {
           int Count;
           PQListNode *ItemList;
        } PriorityQueue;
```

Implementing Priority Queues Using Sorted Linked Lists

```
/* This is the file PQImplementation.c */

#include <stdio.h>
#include <stdlib.h>
#include "PQInterface.h"

/* Now we give all the details of the functions */
/* declared in the interface file together with */
/* local private functions.                     */

void Initialize(PriorityQueue *PQ)
{
   PQ->Count=0;
   PQ->ItemList=NULL;
}
```

Implementing Priority Queues Using Sorted Linked Lists (cont'd)

```
int Empty(PriorityQueue *PQ)
{
   return(PQ->Count==0);
}


int Full(PriorityQueue *PQ)
{
   return(PQ->Count==MAXCOUNT);
}
```

Implementing Priority Queues Using Sorted Linked Lists (cont'd)

```
PQListNode *SortedInsert(PQItem Item, PQListNode *P)
{
   PQListNode *N;

   if ((P==NULL)||(Item >=P->NodeItem)){
      N=(PQListNode *)malloc(sizeof(PQListNode));
      N->NodeItem=Item;
      N->Link=P;
      return(N);
   } else {
      P->Link=SortedInsert(Item, P->Link);
      return(P);
   }
}
```

Implementing Priority Queues Using Sorted Linked Lists (cont'd)

```
void Insert(PQItem Item, PriorityQueue *PQ)
{
    if (!Full(PQ)){
      PQ->Count++;
      PQ->ItemList=SortedInsert(Item, PQ->ItemList);
    }
}
```

Functions `Insert` and `SortedInsert`

- The function `Insert` keeps the elements of the list in **decreasing order** (the first item has the highest priority).
- The function `Insert` calls `SortedInsert` for doing the actual insertion.
- `SortedInsert` has three cases to consider:
  - If the `ItemList` of `PQ` is empty.
  - If the new item has priority greater than or equal the priority of the first item on `ItemList`.
  - If the new item has priority less than that of the first item on `ItemList`. In this case the function is called recursively on the tail of the list.

Implementing Priority Queues Using Sorted Linked Lists (cont'd)

```
PQItem Remove(PriorityQueue *PQ)
{
    PQItem temp;
    if (!Empty(PQ)){
      temp=PQ->ItemList->NodeItem;
      PQ->ItemList=PQ->ItemList->Link;
      PQ->Count--;
      return(temp);
    }
}
```

Function `Remove`

- The function `Remove` simply deletes the item in the first node of the linked list representing `PQ` (this is the item with highest priority) and returns the value of its field `NodeItem`.

The Priority Queue Data Types

In the **unsorted array case**, the file `PQTypes.h` can be defined as follows:

```
#define MAXCOUNT 10

typedef int PQItem;

typedef PQItem PQArray[MAXCOUNT];

typedef struct {
           int Count;
           PQArray ItemArray;
         } PriorityQueue;
```

Implementing Priority Queues Using Unsorted Arrays

```
/* This is the file PQImplementation.c */

#include <stdio.h>
#include "PQInterface.h"

/* Now we give all the details of the functions */
/* declared in the interface file together with */
/* local private functions.                     */

void Initialize(PriorityQueue *PQ)
{
   PQ->Count=0;
}
```

Implementing Priority Queues Using Unsorted Arrays (cont'd)

```
int Empty(PriorityQueue *PQ)
{
   return(PQ->Count==0);
}


int Full(PriorityQueue *PQ)
{
   return(PQ->Count==MAXCOUNT);
}
```

Implementing Priority Queues Using Unsorted Arrays (cont'd)

```
void Insert(PQItem Item, PriorityQueue *PQ)
{
    if (!Full(PQ)) {
        PQ->ItemArray[PQ->Count]=Item;
        PQ->Count++;
    }
}
```

Function `Insert`

- The function `Insert` simply appends the new item to the end of array `ItemArray` of `PQ`.

Implementing Priority Queues Using Unsorted Arrays (cont'd)

```
PQItem Remove(PriorityQueue *PQ)
{
   int i;
   int MaxIndex;
   PQItem MaxItem;

   if (!Empty(PQ)){
      MaxItem=PQ->ItemArray[0];
      MaxIndex=0;
      for (i=1; i<PQ->Count; ++i){
         if (PQ->ItemArray[i] > MaxItem){
            MaxItem=PQ->ItemArray[i];
            MaxIndex=i;
         }
      }
      PQ->Count--;
      PQ->ItemArray[MaxIndex]=PQ->ItemArray[PQ->Count];
      return(MaxItem);
   }
}
```

Function `Remove`

- In the function `Remove`, we first find the item with highest priority. Then, we save it in a temporary variable (`MaxItem`), we delete it from the array `ItemArray` and move the last item of the array to its position. Then, we return the item of the highest priority.

Interface Header Files

- Note that the module interface header file `PQInterface.h` is included in two important but distinct places:
  - At the beginning of the **implementation files** that define the hidden representation of the externally accessed module services.
  - At the beginning of **programs** that need to gain access to the external module services defined in the interface file.

Separate Compilation

- We can compile the module and the client program **separately:**

```
gcc -c PQImplementation.c -o PQ.o
gcc -c sorting.c -o sorting.o
gcc PQ.o sorting.o –o program.exe
```

With the first two commands, we compile the C files to produce **object files**. Then, the object files are **linked** to produce the final executable.

Information Hiding Revisited

- Let us revisit the sorting program we wrote earlier and consider the new `printf` statement.

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

typedef PQItem SortingArray[MAXCOUNT];
/* Note: MAXCOUNT is 10 */

void PriorityQueueSort(SortingArray A)
{
   int i;
   PriorityQueue PQ;

   Initialize(&PQ);
   for (i=0; i<MAXCOUNT; ++i) Insert(A[i], &PQ);
   printf("The queue contains %d elements\n",PQ.Count);
   for (i=MAXCOUNT-1; i>=0; --i) A[i]=Remove(&PQ);
}
```

Information Hiding Revisited (cont'd)

- This `printf` statement accesses the `Count` field of the priority queue `PQ`. Therefore, the previous module organization **has not achieved information hiding** as nicely as we would want it.
- We can live with that deficiency or try to address it. How?

Another Example: Complex Number Arithmetic

- A **complex number** is an expression $a + bi$ where a and b are reals.
- a is called the **real part** and b the **imaginary part.**
- $i = \sqrt{-1}$ is the **imaginary unit**. It follows that $i^2 = -1$.
- To multiply complex numbers, we follow the usual algebraic rules.

Examples

- $(a + bi)(c + di) = ac + bci + adi + bdi^2 = (ac - bd) + (ad + bc)i$
- $(1 - i)(1 - i) = 1 - i - i + i^2 = -2i$
- $(1 + i)^4 = 4i^2 = -4$
- $(1 + i)^8 = 16$
- Dividing the two parts of the above equation by $16 = (\sqrt{2})^8$, we find that $(\frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}})^8 = 1$.

Complex Roots of Unity

- In general, there are many complex numbers that evaluate to 1 when raised to a power. These are the **complex roots of unity**.
- For each N, there are exactly N complex numbers z such that $z^N = 1$.
- The numbers $\cos(\frac{2\pi k}{N}) + i \sin(\frac{2\pi k}{N})$ for $k = 0, 1, \cdots, N-1$ can be easily shown to have this property.
- Let us now write a program that computes and outputs these numbers for a given N.

An ADT for Complex Numbers: the Interface

```
/* This is the file COMPLEX.h */

typedef struct complex *Complex;
Complex COMPLEXinit(float, float);
float Re(Complex);
float Im(Complex);
Complex COMPLEXmult(Complex, Complex);
```

Notes

- The interface on the previous slide provides clients with **handles** to complex number objects but does not give any information about the representation.
- The representation is a `struct` that is not specified except for its tag name.

Handles

- We use the term **handle** to describe a reference to an abstract object.

- Our goal is to give client programs handles to abstract objects that can be used in assignment statements and as arguments and return values of functions in the same way as built-in data types, while hiding the representation of objects from the client program.

Complex Numbers ADT Implementation

```
/* This is the file CImplementation.c */

#include <stdlib.h>
#include "COMPLEX.h"

struct complex { float Re; float Im; };

Complex COMPLEXinit(float Re, float Im)
{ Complex t = malloc(sizeof *t);
  t->Re = Re; t->Im = Im;
  return t;
}

float Re(Complex z)
{ return z->Re; }

float Im(Complex z)
{ return z->Im; }

Complex COMPLEXmult(Complex a, Complex b)
{    return COMPLEXinit(Re(a)*Re(b) - Im(a)*Im(b), Re(a)*Im(b) + Im(a)*Re(b));
}
```

Notes

- The implementation of the interface in the previous program includes **the definition of structure** `complex` (which is hidden from the clients) as well as **the implementation of the functions** provided by the interface.
- Objects are pointers to structures, so we dereference the pointer to refer to the fields.

Client Program

```
/* Computes the N complex roots of unity for given N */
/* This is file roots-of-unity.c */


#include <stdio.h>
#include <math.h>
#include "COMPLEX.h"
#define PI 3.141592625

main(int argc, char *argv[])
{
  int i, j, N = atoi(argv[1]);
  Complex t, x;
  printf("%dth complex roots of unity\n", N);
  for (i = 0; i < N; i++)
    {
      float r = 2.0*PI*i/N;
      t = COMPLEXinit(cos(r), sin(r));
      printf("%2d %6.3f %6.3f ", i, Re(t), Im(t));
      for (x = t, j = 0; j < N-1; j++)
        x = COMPLEXmult(t, x);
      printf("%6.3f %6.3f\n", Re(x), Im(x));
    }
}
```

Notes

- The client program outputs the powers of unity one by one, together with a verification that they are indeed such powers. To verify this, raising to a power is implemented by multiplication.

Notes

- In this case, we can see that the exact representation of a complex number is hidden from the client program.
- The client program can refer to the real and the imaginary part of a number **only by using the functions** `Re` and `Im` provided by the interface.

Command Line Arguments

- `argc` (argument count) is the number of command line arguments.
- `argv` (argument vector) is pointer to an array of character strings that contain the arguments, one per string.
- By convention, `argv[0]` is the name by which the program was invoked so argc is at least 1.
- In the previous program `argv[1]` contains the value of `N`.

Separate Compilation

- We compile the module and the client program separately:

```
gcc -c CImplementation.c -o CI.o
gcc -c roots-of-unity.c -o roots-of-unity.o
gcc CI.o roots-of-unity.o –o program.exe -lm
```

With the first two commands we compile the C files to produce object files. Then the object files are linked to produce the final executable. Notice that we have to use the option `–lm` to link the math library.

Exercise

- Revisit the ADT priority queue and define a better interface and its implementation so that we have information hiding.

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.

  Chapter 4.
- Robert Sedgewick. Αλγόριθμοι σε C.

  Κεφ. 4