EO109
程式語言 (Programming Language)
Semester 102-2

白小明
Jonathon David White
元智大学光电系
R70740, R70723
WhiteJD@XiaoTu.com

修改:中華民國 103 年 5 月 17 日 11 時 39 分 45 秒
Modified AD14 年 5 月 17 日 (Rev 157)

“To Him Who Is Above And Beyond All”

Name	
Student ID	
Class ID	
Cell Phone	
Email	

Table of Contents

0. Introduction

0.1 Facilitators

a. Lecturer: 白小明 小明白

1. Background: http://www.xiaotu.com/whitejd/per/index.htm

Jonathon David White was born in Oakville, Canada but has since lived in many other countries. Even during his undergraduate days at McMaster University, he already had a cosmopolitan outlook on life, being active in the Chinese Christian Fellowship. After obtaining his Ph.D., also from McMaster University, he worked and taught in China, Japan, and Taiwan – where he met and married Wu Xiuman – and then Malaysia at Multimedia University. After 4 years (1999-2003) in the Faculty of Engineering and Technology at the Melaka campus of Multimedia University, he moved with his family to Taiwan. He is now Associate Professor at Yuan Ze University. He and his wife have two daughters, Ai-en (Charity Grace) and Liang-En (Ruth Ann) as well as two sons, You-en (Johann Donald) and Li-En (Leon Joshua). Dr. White's experience in programming has largely been self-taught on a "need-to-know" basis. His introduction to ANSI-C came in 1994, when he took a position in the Ocean Remote Sensing Institute in Qingdao, China. Upon arrival, he was given a book introducing ANSI-C (in Chinese) and told to interface a computer, laser and detector – allowing him to simultaneously learn ANSI-C and Chinese! This "need-to-know" has resulted in the the method of teaching of this course.

2. Family: 爱有力量 https://www.youtube.com/watch?v=G1h9AhUh7o8
3. Research: http://www.xiaotu.com
4. Email: whitejd@xiaotu.com
5. Calendar: http://www.xiaotu.com
6. Office: R70740, R70723 & Lab
7. Office Hours: Tuesdays and Thursdays, 1 PM to 5 PM

b. Teaching Assistants

1. Kevin: Vietnamese Ph.D. student R70740
2. Aray: Taiwanese Ph.D. student R70740

0.2 Respect

a. Classroom Expectations

1. Arrive on Time (after attendance deemed absent)
2. Listen to Lectures
3. Ask Questions (bonus marks)
4. Listen to fellow students
5. Food and Drinks are OK in the classroom
6. Do not leave garbage in classroom
7. During class: (as this distracts other students)
   i. No FACEBOOK,
   ii. No computer games
   iii. No checking email
   iv. No videos

Students disobeying rules will be asked to leave the classroom. If cited more than three (3) times, student will be asked to drop the course.

b. Rules for the Computer Room 電腦教室使用規定

1. 上課注意事項：
   i. 準時到教室，遲到禁止進入教室。
   ii. 在教室裡請勿飲食，食物和飲料禁止帶進室內。
   iii. 每位同學上課都有固定位置，點名前請勿隨意更換位子。

iv. 請勿隨意更動教室內電腦設定。

2. 下課注意要點：
   i. 請將垃圾帶走，丟在安全門外的垃圾桶。
   ii. 請將座椅歸回原本的位置。
   iii. 每個禮拜會安排值日生在課後檢查教室，請務必配合。
   iv. 有違反規定的將登記扣分
   v. 以上如有不清楚的部份，請找老師或助教協助

0.3 Course Overview

This course is the second in a series of three courses for Optics students dealing with computer programming. The goals for this first course are twofold. First, for this second course is to have students understand the research methodology involved in formulating a problem, diagramming the solution and then writing ANSI-C code to aid in the solution of the problem.

The teaching format is lectures followed by small groups(3-4 students) completing a worksheet with help from the teacher and TAs.

Table 1: Key Topics in this course

Topic	題目
Problem formation and UML diagrams	
Simple ANSI-C program	
Basic Program with Functions and Variables (linear flow)	
Using VP diagrams to understand code	
Program Flow	
Arrays and Records	

0.4 Textbooks/References

We will be using selected chapters from the following two textbooks in this course.

1. Ian Chai and Jonathon White, *Structuring Data and Building Algorithms: An ANSI-C Based Approach*, McGraw-Hill (@ Caves, Contact: Tel : 02-23113000#212 / Fax : 02-2388-8822 at McGraw-Hill) ISBN: 978-0071271882 Chapters 1, 7, 11, 12 (in eo109: 1, 2, 7)

0.5 Key Websites

1. http://www.xiaotu.com/tea/yzueo109.htm (Animations for this class)
2. http://www.sdba.info (Textbook Animations for FSA and Turing Machines)

The first textbook will be used for all three courses in computer science offered by our department. Key concepts are covered in the animations and view graphs

0.6 Course Delivery and Milestones

For this course the progress of students is monitored through a series of milestones. Figure 1 shows the topics to be studied in this course and their relationship, along with the key milestones in terms of a modified UML diagram. In this diagram milestones are marked by diamonds.

Fig. 1 UML illustration of tested study milestones for this course.

.

0.7 Grading

Table 2: Milestones and Their Weight for Midterm and Final Assessments

項目編號	項目名稱 Milestone	期中評量權重 Midterm	學期總成績權重 Final
1	Simple ANSI-C program : creating and debugging	8%	4%

項目編號	項目名稱 Milestone	期中評量權重 Midterm	學期總成績權重 Final
2	Problem formation and UML diagrams	24%	12%
3	Basic Program with Functions (linear flow)	48%	24%
4	Using VP diagrams to understand code	0%	16%
5	Program Flow	0%	12%
6	Arrays and Records	0%	12%
0	Attendance, Tutorials and Small Group work	20%	20%
0	BONUS: Successful Group Leaders, Pointing out errors	max 5%	max 5%
	TOTAL	105%	105%

As can be seen in Fig. 2 and Table 2, grading is based both on (1) attendance and participation in small groups, (2) performance on six key milestones and (3) bonus activities. Bonus marks are available for pointing out errors and mistakes in the teacher's lecture materials. Each mistake will give the first student who points it out an additional 1 point. Each student can earn a maximum of 5 points for finding errors in the teacher's lectures. The percentage each milestone contributes to the final mark is directly proportional to the number of hours assigned this topic with each hour of study being awarded 2 points in the final evaluation. For example, since six (6) hours are spent studying problem formation and UML diagrams, this milestone is worth twelve (12) points in the final evaluation and twice that on the midterm evaluation. Unlike other courses, each milestone is evaluated on a Pass/Fail basis and each student can try three (3) times to pass the milestone. If one passes the milestone at one's first attempt, one receives the full point score for the milestone. If, however; requires a second attempt to pass the milestone, then only 85% of the marks assigned that milestone will be awarded. For third attempt only 75% of the maximum marks are given. Detailed requirements for each milestone are listed below:

1. Compile/Run 1 Function. Student must be able to write a simple program in jEdit (does not need to do anything) and then compile, link and run the program. This includes downloading and installing the compiler and editor on your own computer (if you do not have a computer, you can request alternate examination) and demonstrating the use of the debugger program.
2. UML. Student must be able to look a problem and write a possible solution to the problem using Universal Modelling Language Activity Symbols. Knowledge of start symbol, activity symbol, flow symbol, comments, splitting and parallel processing is required.
3. Basic Program. Student should be able to write a simple program making use of functions (written by himself, others and from the standard library). Sample question: write 2 functions to calculate the calculate the volume and surface area of the cylinder. Allow the user to input the radius and height of the cylinder. Call the two functions for calculation and print the volume and surface area of an arbitrary cylinder input by the user using standard library functions fprintf()
4. Code to VP-Diagram. Student should be able to draw VP diagrams for a short piece of ANSI-C code that makes use of functions, pointers and variables.
5. Program flow. Student should be able to write a function that combines conditional, repetition and sequential flow.
6. Arrays and Records. Student should be able to write programs that make use of arrays.

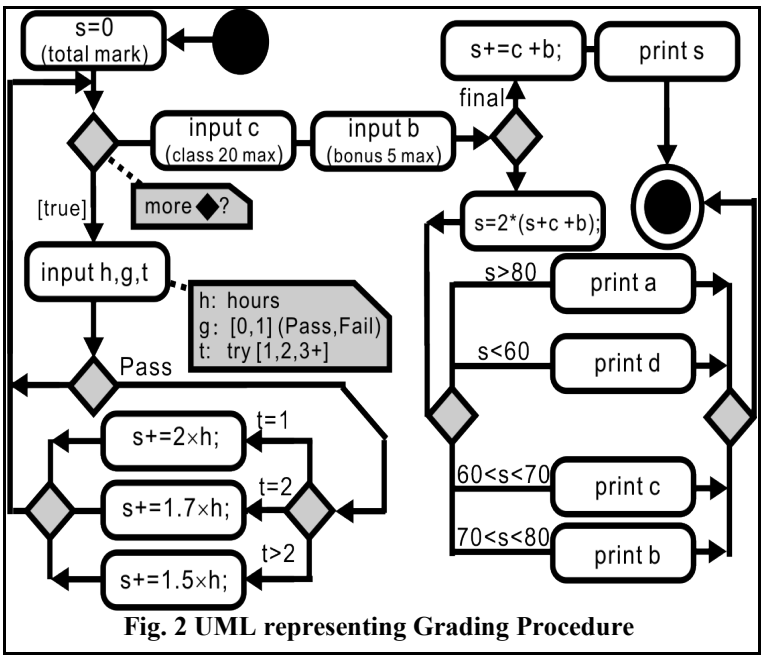


Fig. 2 UML representing Grading Procedure

0.8 Calendar

The times on this calendar are tentative and may be changed.

Table 3: Course Plan and time table

Class	Topic	Wk	Date
1	Welcome, Course Introduction	1	02.18 @ 08:00
2	Problem Introduction – falling objects, optics, square root	1	02.18 @ 10:00
3	(review) 1st program: problem formation, Using jEdit	2	02.25 @ 08:00
4♦	(review) 1st program: Compiling & Linking, Debugging	2	02.25 @ 10:00
5	UML, Functions & Commented Code	3	
6	UML, Functions & Commented Code	3	
♦	Retry Milestone #1: 1st program	3	03.04 @ 12:00
7	Introduction to Variables	4	
8	Introduction to Variables	4	
9a♦	Milestone 2: UML diagrams	5	
9b	Applying Variables & Functions: Falling Object	5	
10	Applying Variables & Functions: Falling Object	5	
11	Applying Variables & Functions: Optics	6	
12	Applying Variables & Functions: Square Root	6	
13	Application: Using Variables and Functions	6	
14	Application: Using Variables and Functions	6	
15a♦	Milestone 3: Basic Program	7	
15b	Pointers – an introduction	7	

Class	Topic	Wk	Date
16	Pointers – an introduction	7	
17	Pointers – usage	8	
18	Pointers – application & Introduction to Program Flow	8	
♦a	Retry Milestone 1: Simple Program (3rd and last attempt)	9	04.15 @ 8:00
♦b	Retry Milestone 2: UML (2nd attempt)	9	04.15 @ 9:00
♦c	Retry Milestone 3: Functions & Variables(2nd attempt)	9	04.15 @ 11:00
19a♦	Milestone 4: Code to VP-diagrams	10	
19b	Program Flow	10	
20	Program Flow	10	
♦	Retry Milestone 2: UML (3rd and last attempt)	10	04.22 @ 12:00
21	Program Flow	11	
22	Arrays	11	
♦	Retry Milestone 3: Functions & Variables(3rd & last attempt)	11	04.22 @ 12:00
23♦	Milestone 5: Program Flow Arrays	12	
24	Arrays	12	
25	Arrays – Application	13	
26	Introduction to Records	13	
27♦	Milestone 6: Arrays	14	
28♦	Retry Milestone 4, 5, 6: Times to be announced	14	
29♦	Retry Milestone 4, 5, 6: Times to be announced	15	
23♦	Retry Milestone 4, 5, 6: Times to be announced	15	

0.9 Detailed Lecture Plan / Teaching Schedule with References (see online)

1. Defining the Problem – Falling Objects, Optical Ray Tracing, Root Finding

2. Organizing a Solution – Algorithms, UML Activity Diagrams, Functions & Commented Code

Fig. {*muffin}* Snickerdoodle Muffin*

2.1 Algorithms

a. Definition

An algorithm describes the steps that one needs to take to take in order to perform a certain task or computation. On the one hand, the same algorithm may be expressed in many different languages and still be the same algorithm even though it may look quite different. On the other hand, different algorithms can be used to perform the same task.

More formally, an algorithm is defined as:

> A finite sequence of steps,
> - each step consisting of a number of operations,
> - each operation
>
> 1. is rigorously defined and unambiguous
> 2. can be executed by a machine

b. Example: An algorithm to make muffins

A recipe is an example of an algorithm that describes how to prepare a specific dish or meal. Consider for example the following algorithm that describes how one makes muffins*.

1. Prepare ingredients: 2 cups flour, 1 tsp baking powder. 5 tbsp milk powder, 1 egg, tbsp olive oil, 1 cup water
2. Turn oven on to 250 C.
3. Mix wet ingredients (water,olive oil, egg)
4. Mix dry ingredients (milk powder,flour,baking powder)
5. Pour wet ingredients into dry ingredients.
6. *(Option: Add 1 cup of fruit(i.e. blueberries) into batter.)*
7. Mix leaving a few lumps in the batter
8. Pour batter into muffin tray.
9. Bake in oven for 20 minutes
10. *If brown (finished cooking), GOTO Step 12*
11. *Bake for 1 more minute. GOTO Step 10*
12. Take out of Oven

This recipe explains the steps one follows to make muffins. It provides all the information that one needs to know to be successful. The new cook does not need to rediscover anything. Ignoring steps 6, 10, and 11 (in italics), the flow is seen to be linear – no decisions to make. Step 6 is a conditional or option – a decision needs to be made about whether to add berries to the muffin. Steps 10 and 11 indicate repetition: they need to be preformed a number of times until a condition is met. As a final note, notice that the order in which steps 3 and 4 are completed is not important – in fact, they could be done in parallel. In the following section we will introduce UML activity diagrams that allow us to illustrate diagrammatically this algorithm.

---

* http://eatathomecooks.com/2010/05/snickerdoodle-muffins.html (Downloaded picture on 2014.04.16)

2.2 UML Activity Diagrams

a. Overview

Expressing your algorithm clearly before starting to write computer code is crucial for creating easy to understand, well structured code. In order to help you to do this, a standard, called Universal Modelling Language, UML for short, has been developed to help you learn to think before you start to code. (In the past we have used UML state diagrams to respresent FSAs, now we will use UML activity diagrams to represent algorithms. If you have done programming before, you may have made use of a flowchart to represent an algorithm. The UML Activity diagram replaces a traditional flowchart.

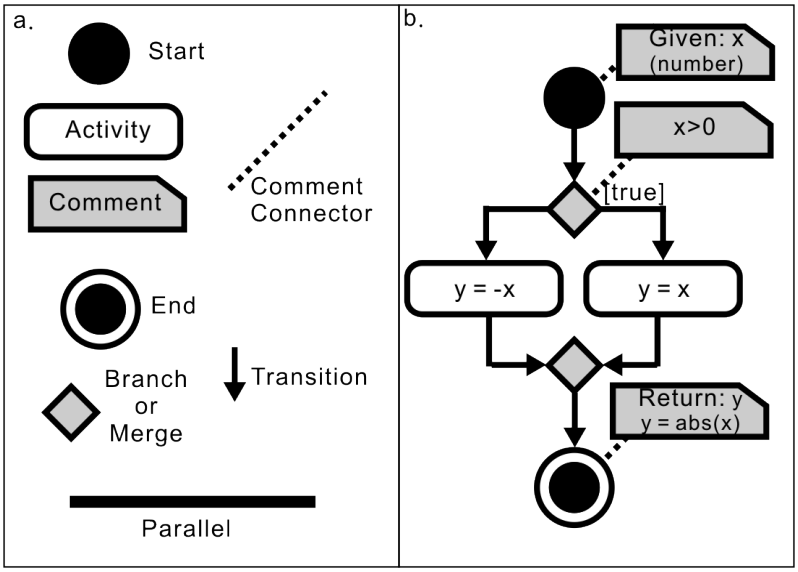


Fig. *{UML}* (a) Key symbols used to describe algorithms using UML activity diagrams. (b) Putting together the symbols to describe and algorithm to take the absolute value of a number.

Figure *{UML}* summarizes the key symbols that are used in UML activity diagrams. These diagrams are used to help us to show the steps in an algorithm. Fig. *{UML}* (a) identifies the symbols while Fig. *{UML}* (b) uses these symbols to describe the process are placed together to describe an algorithm to take the absolute value of a arbitrary number.

UML activity diagrams generally make use of five different symbols to represent Start, End, Activity (i.e. verb), Comment, Branch or Merge and three different types of lines to represent Transitions (solid lines with arrows), connect comments, and finally indicate Parallel processes. Within a given UML diagram, there should only be one start symbol. Activities are conducted in the order they appear in the diagrams. In the case that order is not important for two activities, then one can use the Parallel bar to indicate this. Fig. *{UML}* (b) shows how one can diagram an algorithm that returns the absolute value of a number. The the algorithm starts with a number (x) (indicated in comment box as a start condition.). Next a decision is made: if x>0 then we take the right path. If x<0 we take the left

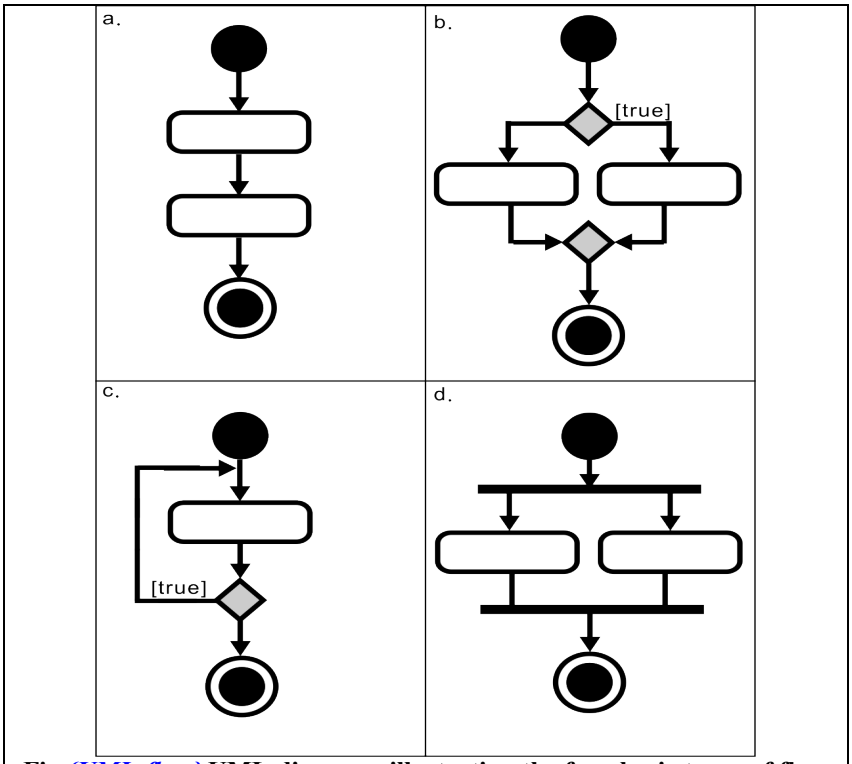


Fig *{UML-flow}* UML diagrams illustrating the four basic types of flow (a) linear-sequential (b) conditional, (c) repetition (d) parallel processing.

path. In the case that we take the right path, we just store the value of x in the variable y. In the case that we take the left path, we negate x and store the result in y. The paths then join back together (merge symbol) again and the algorithm ends. Note that if we take the right path, then we do not take the right path.

Figure *{UML-flow}* illustrates the four types of flow that can be implemented in a program. All algorithms can be expressed in terms of a combination of these flow structures. In Fig.*{UML-flow}*(a) linear-sequential flow is illustrated. In this type of flow, the activity in the upper box is first completed and then activity in the next box can be started. In (b) conditional flow is illustrated. Based on the the value of a variable or some condition either the right path is taken or the left path is taken. Both paths are never taken. Note that the diamond at beginning of the condition represents splitting of program flow while the diamond at the end represents a merging of the flow streams. In Figure *{UML-flow}* (c) illustrates repetition. In this type of flow, a given activity is repeated until some condition is met. Note that there is only one diamond representing the test condition and that the transition line pointing back ends on a transition line (not an activity box) Finally in (d) a specialized type of flow is illustrated: "parallel" processing. In this case, both activities are completed but the order in which they are completed is not important. Thus they can be done in parallel. In contrast to sequential processing, there is no order for activities. In contrast to conditional processing, both branches are taken. Parallel processing is denoted by a thick lines that mark both the beginning and end of the parallel activities.

b. Conditional Flow

In structured programming, conditional flow is generally structured in one of the three ways shown in Fig. *{UML-cond}*. In each case comment boxes can be used to repesent the condition to be tested. If the condition is very simple, then it can just be written on a line leaving the split symbol.

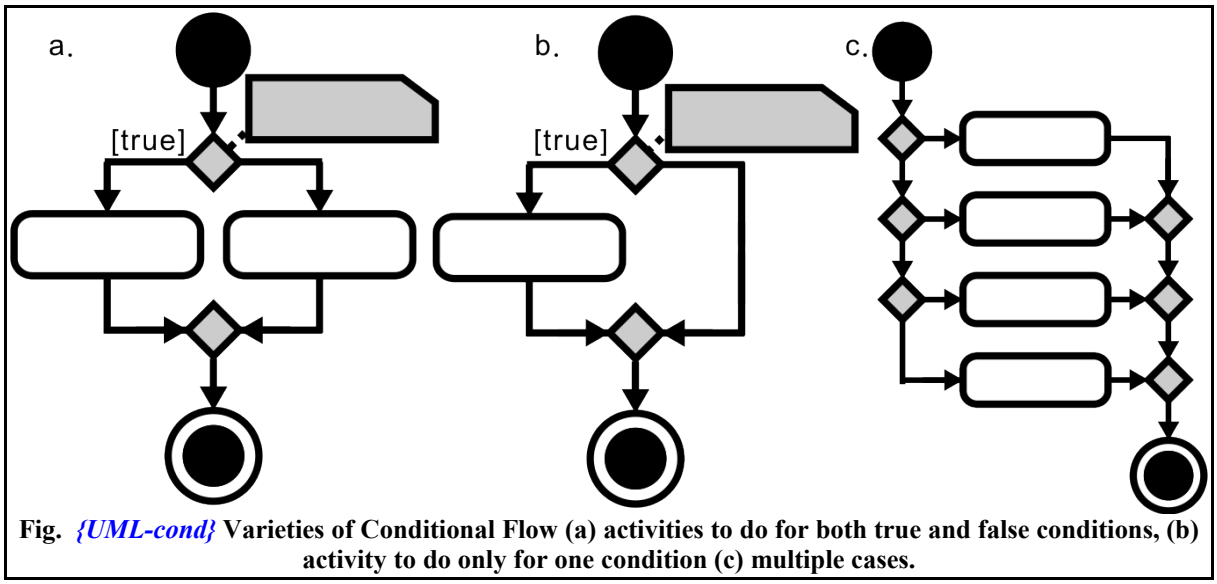


Fig. *{UML-cond}* Varieties of Conditional Flow (a) activities to do for both true and false conditions, (b) activity to do only for one condition (c) multiple cases.

In (a) we illustrate the structure if an action needs to be preformed no matter what the outcome of the check. (b) represents the condition that we only need to preform an action if something is true or false. For example, in our function to calculate absolute value of x, if we didn't want to create a new variable y but rather keep only one variable x, then we need do nothing in the positive case. Finally (c) represents a situation when we do different things in many different cases. For example, a function that based on the maximum education received by an individual preforms a different actions. In this case, on might process data based on whether the individual has graduated from university, high school, primary school or has received no formal education.

c. Repetition

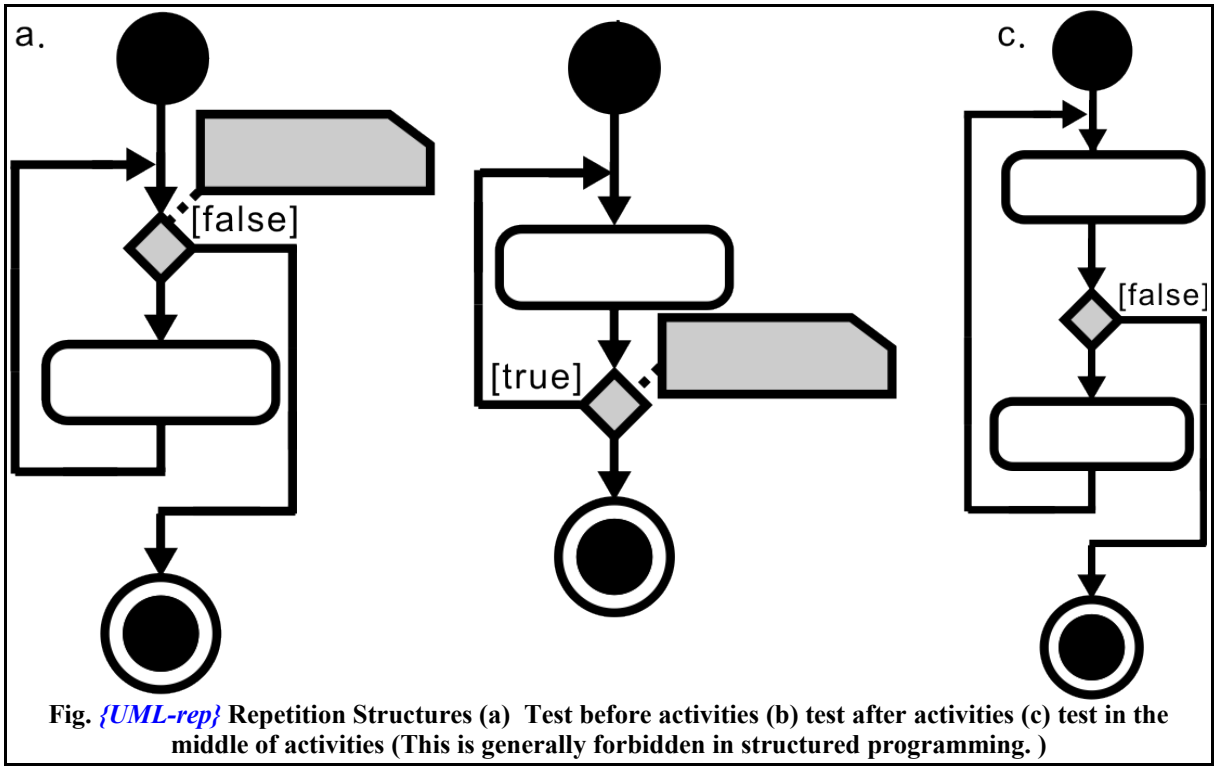


Fig. *{UML-rep}* Repetition Structures (a) Test before activities (b) test after activities (c) test in the middle of activities (This is generally forbidden in structured programming.)

In structured programming, repetition is generally structured in one of the two ways shown in Fig. *{UML-rep}*. Figure *{UML-rep}*(a) represents the case in which the condition is tested before the loop any activity is completed while (b) represents the case that the condition is tested after the activities have been completed one time. The key difference is that, in the first case, it is possible for the activities within the loop never to be executed Finally (c) represents the condition that the looping condition is tested in the middle of the loop. In structured programming this type of structure is forbidden as it makes code difficult to read. Note that the condition, if complicated, is written in a comment box and one transition is labelled as [true] or [false]. Alternatively, if the condition is simple we can write it on one of the transitions lines.

d. Common technical drawing mistakes

Other than logic, there are a number of common technical mistakes that are often made in drawing these types of diagrams. These are illustrated in Fig.*UMLmistake,*namely, arrows not connected to anything (a), skipping the merge icon on conditionals (b), feeding back into an activity box (c), multiple inputs into a merge box (OK in special

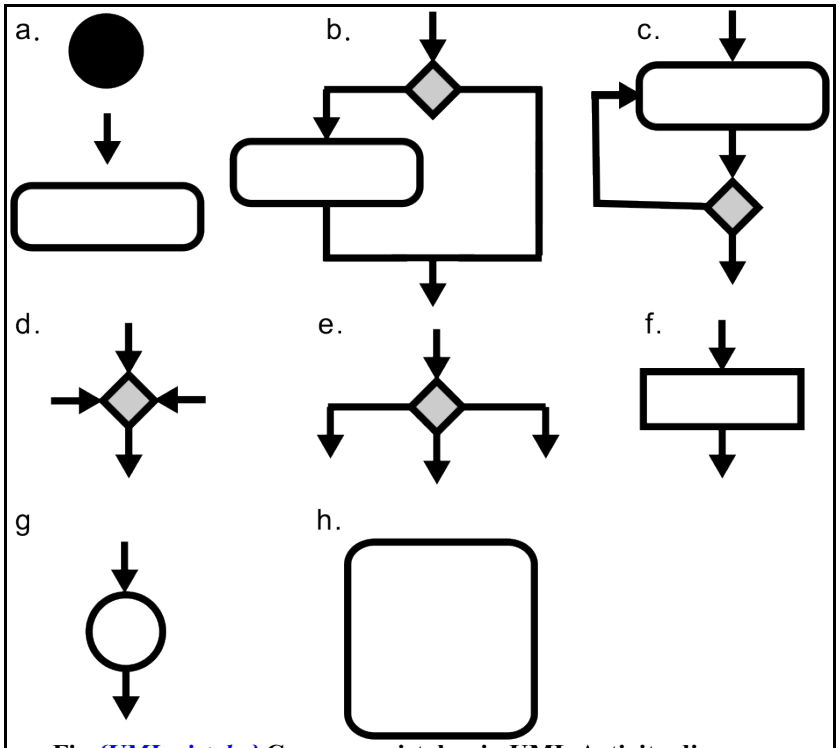


Fig *{UMLmistake}* Common mistakes in UML Activity diagrams

cases) (d) , multiple exits from a split symbol (OK in special cases) (e), non-rounded rectangles (f), and circles (g). Non-rounded rectangles are used in structural UML diagrams (not activity UML diagrams) to represent classes and circles are used in Finite State Diagrams. Finally in (h) we demonstrate the mistake of putting a number of different steps all in one box or using complicated English to describe a process (not machine executable). Each step should have its own box in an activity UML diagram.

2.3 Example: UML for Baking Muffins

As a final example of UML, we will convert our baking recipe to UML Activity diagrams, For simplicity in the example, just the step number is written in the activity boxes or comment box. The diagram makes it quite obvious that, at least at one point in time, at least three tasks can be done simultaneously. For example, I can have my youngest daughter, Liang-En doing step 3, Li-En doing step 4 and I can send my oldest daughter Aien warming up the oven.

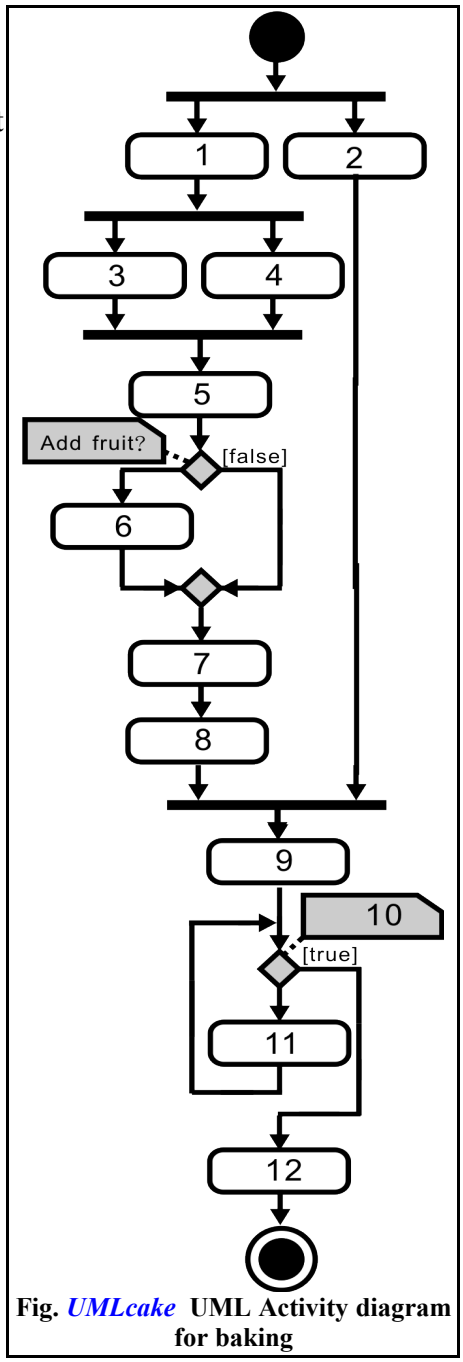


Fig. *UMLcake* UML Activity diagram for baking

2.4 Functions

3. Variables

3.1 Introduction (I Chai, pg 3-4)

3.2 Containers with Names (I Chai, Pg 4-6)

3.3 Fixed-Point vs Floating Point Containers (I Chai, pg 6-15)

3.4 Declaration, Initialization, Usage

3.5 Review Example: From Problem to Final Solution

a. Problem Background and Statement

A number of old bombs have been unearthed in Gipuzko(cf Fig. {bomb}). These bombs have the shape of a hollow sphere. Your coworkers have compiled a list of the bombs found along with their weight, and inner and outer diameters. Write a program to calculate the density of the material used to make the bombs. (From the density, one can then try to guess the material used to make the bombs.)

Fig. {bomb} A bomb from Gipuzko
http://bertan.gipuzkoakultura.net/img/18/grandes/130.jpg

b. Think

The first step is think about how to solve the problem, compile the equations needed (do a literature search if necessary) and attempt to solve the problem for a simple case.

c. Summarize and write as mathematical variables the information we have received:

d_{outer} : outside diameter of the spherical bomb

d_{inner} : inside diameter of the hollow shell

m : mass of the bomb

d. Compile (search for in the literature) the necessary equations.

Density (p) is related to mass (m) and volume (V) of the material by

$$p(m,V) = m/V \qquad (1)$$

From basic geometry, the volume of a solid sphere can be related to its diameter (d) by the expression as:

$$V_{sphere}(d) = \pi d^3/6 \qquad (2)$$

The volume material of the hollow shell can be expressed as the difference in volumes of an outer and inner sphere:

$$V_{hollowShell}(V_{outer}, V_{inner}) = V_{outer} - V_{inner} = V(d_{outer}) - V(d_{inner}) \qquad (3)$$

e. Do a simple hand calculation

Having assembled the required equations, we can then make a sample calculation with some simple numbers. For example,

$d_{outer} = 20$ cm

$d_{inner} = 10$ cm

m = 30 kg

First we calculate the volumes of the outer sphere and the inner hollow region:

$$V_{outer}(d_{outer}) = \pi d_{outer}^3/6 = = \pi\, 20^3/6 = 4000\pi/3$$

$$V_{inner}(d_{inner}) = \pi d_{inner}^3/6 = = \pi\, 10^3/6 = 1000\pi/6$$

From this we can estimate the total volume of material in the bomb:

$$V_{hollowShell}(V_{outer}, V_{inner}) = V_{outer} - V_{inner} = 4\pi/3 - \pi/6 = 7000\pi/6 \sim 3660\ cm^3$$

and finally the density of the material used to make the bomb casing

$p(m,V) = m/V = 30000\ g / 3660\ cm^3 \sim 8\ g/cm^3$

We can then compare this density with normal density of materials. From Wikipedia we can look up the densities of a number of materials: (Aluminum, 2.643. Brass, 8.553. Cobalt, 8.8. Copper, 8.9. Gold, 19.32. Ice, 0.897. Iron, 7.86. Lead, 11.37 g/cm^3) Our result is closest to that for Iron and so we conclude that probably this bomb shell is made predominately from Iron with maybe the addition of some Lead.

f. UML diagram

Once we have worked through a sample problem, the next step is to summarize our algorithm using a UML diagram.

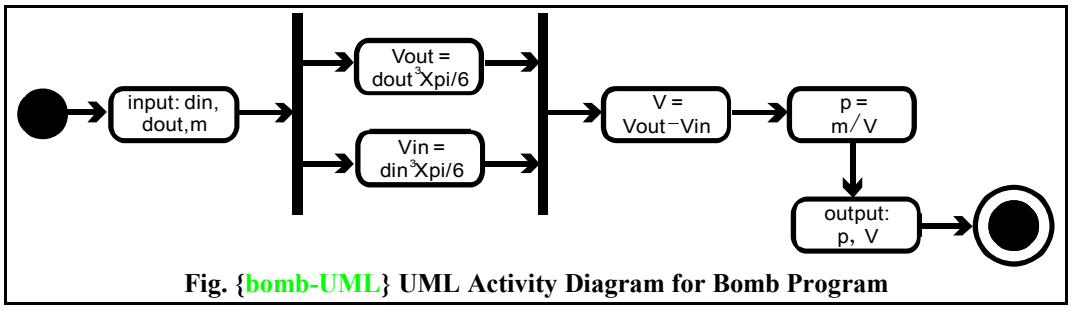


Fig. {bomb-UML} UML Activity Diagram for Bomb Program

In the above diagram, we can see that the first step is to get the required information from the user. Next we calculate both the inner and outer volumes for the spherical shell. Since order is not important, we draw these steps in parallel (note that these both use the same function with different inputs). This is followed by the calculation of the total volume of material in the bomb, followed by the density of the bomb. Note that these steps must follow in order, i.e., we cannot calculate the density of the bomb until we have calculated the total volume of material in the bomb. We cannot calculate the total volume of material until we have first calculated the inner and outer volumes. This is sequential programming.

g. Commented Code

In the commented code, we make the decisions about (1) the functions to use in preforming the calculations, (2) decide whether we write our own functions, use standard library functions or those written by another person and (3) determine the order for parallel operations in the UML diagram.

```
/* bomb1.c : material in bomb shell */
/* (c)2013 J D White */
/* proto: vSphere */
/* proto: vHollow */
/* proto: density */
int main(void){
   /* jlib/getFloat din,dout,m */
   /* Calc Vin call: vSphere */
   /* Calc Vout call: vSphere */
   /* Calc Vbomb call: vHollow */
   /* Calc p call: density */
   /* stdio/fprintf p,V */
   return(0);
}
/* defn: vSphere
   in: d
   out: v
   calc: d*d*d*pi/6
```

```
*/
/* defn: density
   in: mass, volume
   out: p
   calc: p = mass/volume
*/
/* defn: vHollow
   in: vOut, vIn
   out: v
   calc: v= vOut - vIn
*/
```

h. ANSI-C Source Code Initial Draft Version (for debugging)

In this step, one converts the commented code into real ANSI-C code remembering to (1) declare, initialize and use all variables, and (2) prototypes, call and define all functions. In addition, one seeks to ensure that the variable names are clear to anyone reading the program.

```
/* bomb1.c : material in bomb shell */
/* (c)2013 J D White */
#include <stdio.h>     /* for fprintf */
#include "jlib.h"      /* for getFloat */
double vSphere(double diameter);
double density(double mass, double volume);
double vHollow(double vOut, double vIn);
int main(void){
   double dIn,dOut,m; /* input parameters */
   double vIn,vOut,m,v; /* calculated */
   fprintf(stdout,"inner dia");dIn =getFloat();
   fprintf(stdout,"outer dia");dOut=getFloat();
   fprintf(stdout,"mass");     m   =getFloat();
   vIn=vSphere(dIn);
   vOut=vSphere(dOut);
   v=vHollow(vOut,vIn);
   p=density(m,v);
   fprintf(stdout,"p=%lf",p);
   return(0);
}
double vSphere(double d){
   double v;
   v=d*d*d*pi/6;
   return(v);
}
double density(double m, double v){
   double p;
   p=m/v;
   return(p);
}
double vHollow(double vOut, double vIn){
   double v;
   v= vOut - vIn;
   return(v);
```

```
}
```

i. Compile and resolve all warning and error messages

The program file is first compiled, with flags to check of ANSI-C violations and to display all Warning (-Wall) messages.

```
gcc -c -ansi -Wall -g bomb1.c
```

After the program can compile without warning messages, one can then move onto the next step.

j. Link

The program code is linked with the functions that have been called from the standard input and output library (stdio), i.e. fprintf(), and that were written by others, i.e. getFloat() from jlib.

```
gcc -o b.exe bomb1.o jlib.o
```

k. Run

Finally we run the completed code with our test data to verify that the answers are as we expect.

```
b
inner dia float >> 10.0
outer dia float >> 20.0
mass float >> 30.0
p=8.18926
```

l. Debug

If our program does not give the expected answers then we need to look at the code step-by-step to try to find logic or coding errors. We do this by calling the debugger program with our file as the input data

```
gdb b.exe
```

One then steps through the code line-by-line to seek to find any errors. For the above code we might start as follows:

```
b main
r
display dIn
display dOut
display m
display vIn
display vOut
display m
display v
n
```

This way we can step through the program and find which variable is not taking the value expected and thus isolate the problem.

m. ANSI-C Source Code Final Version

In this step, we try to compress the code and make it simpler by removing unnecessary variables and lines.

```
/* bomb1.c : material in bomb shell */
/* (c)2013 J D White */
#include <stdio.h>     /* for fprintf */
#include "jlib.h"      /* for getFloat */
```

```
double vSphere(double diameter);
double density(double mass, double volume);
double vHollow(double vOut, double vIn);
int main(void){
   double dIn,dOut,m; /* input parameters */
   fprintf(stdout,"inner dia");dIn =getFloat();
   fprintf(stdout,"outer dia");dOut=getFloat();
   fprintf(stdout,"mass");     m   =getFloat();
   fprintf(stdout,"p=%lf",density(m,
       vHollow(vSphere(dOut),vSphere(dIn))));
   return(0);
}
double vSphere(double d){return(d*d*d*pi/6);}
double density(double m, double v){return(m/v);}
double vHollow(double vOut, double vIn){
   return(vOut - vIn);
}
```

Note that the final version has put all the functions into one line on the main program. If there was any error in the program, this would be quite difficult to debug. Thus our first version is on many lines.

4. Flow

4.1 Introduction

a. Motivation

b. Three Types of Flow in Structured Programming

4.2 Decision

a. Format

b. Testable Conditions

4.3 Repetition

4.4 Summary

In structured programming there are four types of flow: linear, conditional, repetition and parallel flow. In linear flow, execution starts at the beginning and continues line by line. Parallel flow is outside the scope of this course. Condition and repetition are implemented as follows:

a. Conditional Flow

Two structures are used in ANSI-C to implement conditional flow.

1. if/else structure

```
int getSign(int k){
   int sign;
   if(k<0)sign=-1;
   else sign=+1;
   return(sign);
}
```

Depending on the value of k, positive or negative, either the **`if`** block or the **`else`** block is executed.

2. if without else

```
int getAbsoluteValue(int k){
   if(k<0)k=-k;
   return(k);
}
```

In this case, we only need to take action if the input number is negative. If the number is positive then there is nothing to do.

3. Complex if/else conditions.

For more than one instruction, one should use the curley brackets to surround the contents

```
if(a>=0){
   s=sqrt(a);
   fprintf(stdout,"sqrt(%lf) is %lf",a,s);
}else{
   s=sqrt(-a);
   fprintf(stdout,"sqrt(%lf) is i%lf",a,s);
}
```

4. switch statement with multiple cases:

```
switch(nsoln){
   case 0 : fprintf(stdout,"no solution"); break;
   case 1 : fprintf(stdout,"one solution"); break;
   case 2 : fprintf(stdout,"two solutions"); break;
   default : fprintf(stdout,"more than 2 solutins");
```

```
}
```

Alternatively the above code could be implemented as a series of if/else statements, as shown below:

```
if(nsoln==0)fprintf(stdout,"no solution");
else if(nsoln==1)fprintf(stdout,"one solution");
else if(nsoln==2)fprintf(stdout,"two solutions");
else fprintf(stdout,"more than 2 solutions");
```

b. Repetition

Three structures are used in ANSI-C to implement repetition.

1. While (eg. summing numbers input at the keyboard)

```
int k=0, n=0;
k=getFixed();
while(k>0){
   n+=k;
   k=getFixed();
}
```

While loops are usually preferred over for loops for the case in which the number of times a loop excecutes depends on statements within the loop itself.

2. For (eg. summing numbers from 0 to n)

```
int k, s=0,n=10;
for(k=0;k<n;k++){
   s+=k;
}
```

For loops are generally used when the programmer knows in advance how many times the loop will execute. In other words the number of loops is not affected by anything within the repetition structure.

3. Do...while (eg. throw away negative numbers at beginning of file)

```
int k, n=0;
do{
   k=getFixed();
}while(k<0);
```

Similar to a while loop but used when programmer needs the loop to execute at least one time.

5. Pointers

5.1 Declaration, Initialization, Visualization (I Chai, pg 15-26)

5.2 Variables, Pointers and Functions (I Chai pg 27-30)

5.3 Summary and Review Example: Selections using pointers.

a. Problem Statement

You have been hired to prepare a menu based on patient choices and diet restrictions for patients at an old age home. Residents may choose one portion of vegetables and one portion of meat for their evening meal. In addition they may choose an double portion of single item. You have been given the following chart (the real menu contains more items):

Category	Item	Portion Size (pieces)
Vegetable	Peas	12
Vegetable	Carrots	6
Meat	beef	1
Meat	mutton	2

Based on the choices of the resident and the constraints given above, write a program that asks the resident to choose their menu items and prints the total number of pieces of food the resident will receive.

b. Think and a Example Solution

Choose Vegetable=peas,

Choose Meat=mutton,

Choose DoublePortion=meat selection;

Calculate Total items = 12 + 2 +2 = 16

c. UML Activity Diagram

This one can draw.

d. Commented Code

```
int main(void){
   /* set up the variables (do in main) */
   /* get vegetable choice, tell user quantity */
   /* get meat choice, tell user quantity */
   /* get double portion selection */
   /* calculate total items */
}
```

e. ANSI-C code

In order to write to test the code, we will not use conditional flow. Rather for the following program we will write functions that we can later easily modify to allow the resident to enter their personal choices, once we have verified our program runs correctly. Thus for our first version we will force the choices to stay at a default choice and include a comment line to remind us of the next modifications to get the data. Note that for the chooseDoublePortion function it is necessary to use pointers to pointers in order to complete the manipulation.

```
/* menu.c: preparing a menu */
/* (c)2014 APR 14 J D White    */
#include <stdio.h>
int *chooseVegetable(int *peas, int *carrots);
```

```
int *chooseMeat(int *beef, int *mutton);
int **chooseDoublePortion(int **f, int **v, int **m);
int calcItems(int nVegetable,int nMeat, int nDouble);
int main(void){
   int peas=12, carrots=6;
   int beef=1, mutton=2;
   int *veggee=NULL, *meat=NULL;
   int **doubleP=NULL, nitems=0;
   veggee=chooseVegetable(&peas,&carrots);
   meat=chooseMeat(&beef,&mutton);
   doubleP=chooseDoublePortion(&veggee,&meat);
   nitems=calcItems(*veggee,*meat,**doubleP);
   fprintf(stdout,"total of %d items.",nitems);
   return(0);
}
int *chooseVegetable(int *peas, int *carrots){
   int *f=NULL;
   fprintf(stdout,"Choose either %d peas or %d"
      " carrots. (Enter p for peas)",*peas,*carrots);
   f=peas;/* get choice of Veggie */
   return(f);
}
int *chooseMeat(int *beef, int *mutton){
   int *f=NULL;
   fprintf(stdout,"Choose either %d pieces of beef or %d"
      "pieces of mutton. (Enter b for beef)",*beef,*mutton);
   f=beef /* get choice of Meat */
   return(f);
}
int **chooseDoublePortion(int **v, int **m){
   int **d=NULL;
   fprintf(stdout,"Choose a double portion of veggees"
      "or meat. (Enter first letter)");
   d=v; /* get choice of double Portion */
   return(d);
}
int calcItems(int nVegetable,int nMeat, int nDouble){
   int items=nVegetable+nMeat+nDouble;
   return(items);
}
```

f. Creating a VP diagram.

To help us to understand the code as it works, it is helpful for us to make a VP diagrams showing the status of the variables and pointers at different points in the function.

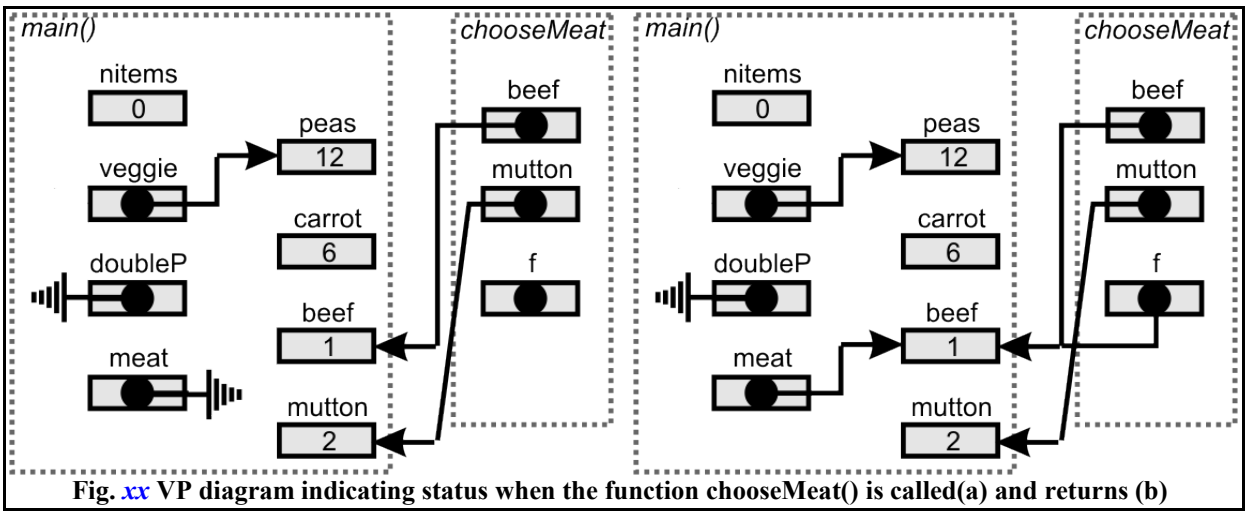


Fig. *xx* VP diagram indicating status when the function chooseMeat() is called(a) and returns (b)

Next we will move down to after choosedoubleportion() returns....

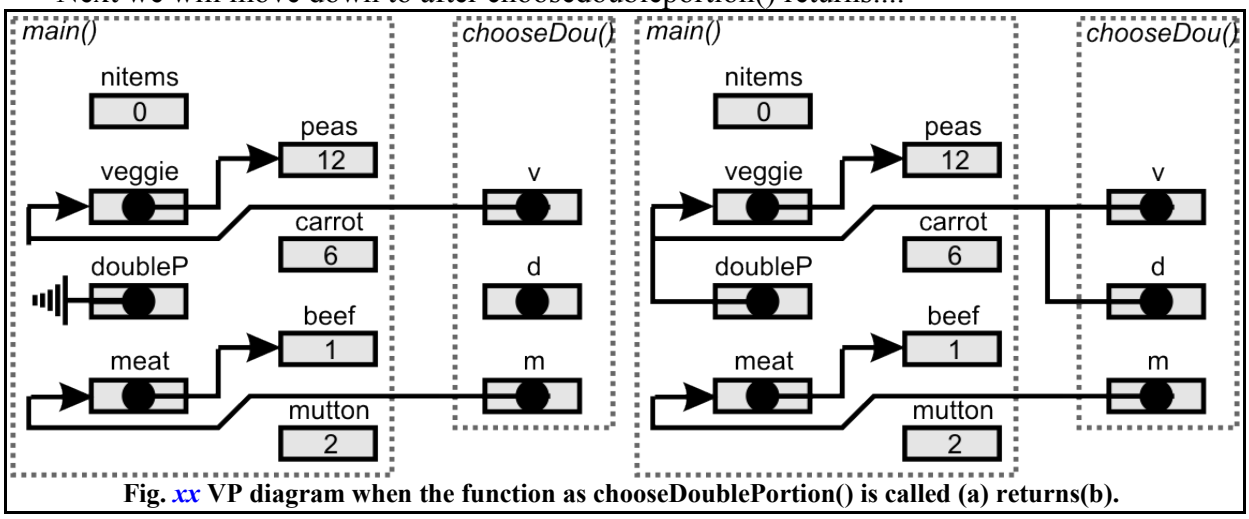


Fig. *xx* VP diagram when the function as chooseDoublePortion() is called (a) returns(b).

Finally lets notice how the calcItem() function works.

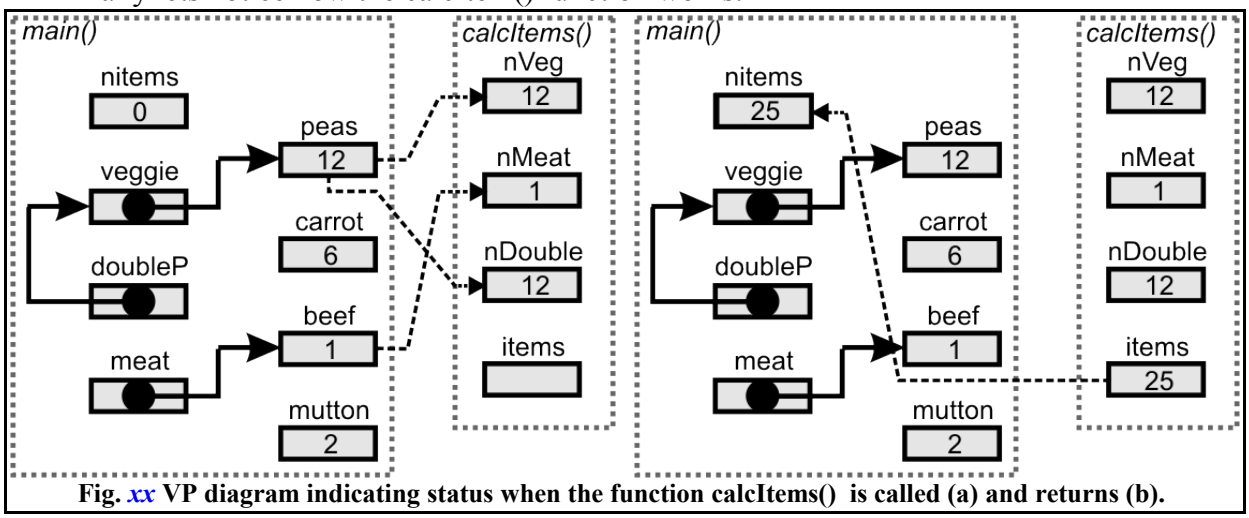


Fig. *xx* VP diagram indicating status when the function calcItems() is called (a) and returns (b).

After looking at these relatively complicated VP diagrams, you must be thinking, "Is there an easier way?" The answer of course is, "Yes". But we will need to wait until we can use arrays of records and strings (an array of characters) or, alternatively a linked list of records and strings.

5.4 Sample Questions

a. Write a function that determines if the roots of a quadratic equation are real or imaginary and then calculates these roots. There are three pieces of information, that the subroutine needs to share with the calling program: nature of the roots (real or imaginary) and the two roots themselves.

6. Arrays

6.1 Motivation: Falling Object Problem

a. Allowing us to write a complete table (well formatted output)

b. Allow input, calculation and output phases of a program to be separated

6.2 Variables, Pointers and Arrays (I Chai, pg 36)

6.3 Declaration: static and dynamic arrays (I Chai, pg 37-42)

6.4 Initialization (I Chai pg 42-45)

6.5 Usage (I Chai pg 46-47)

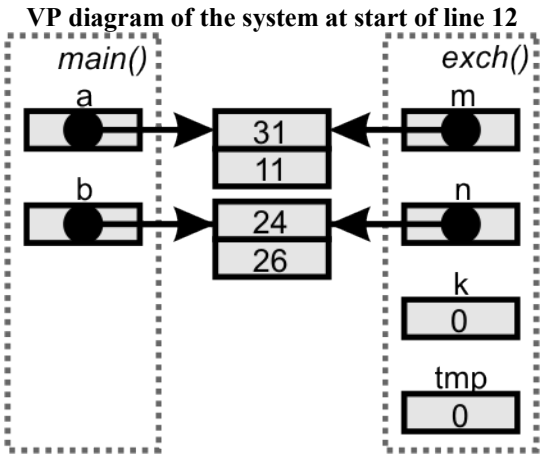
6.6 Notation: Pointer vs Array (I Chai, pg 48)

6.7 Strings: An array of characters (I Chai, pg 48)

6.8 Arrays and Functions

In this example two arrays are passed to a function. The purpose of the function is to swap the contents of the two arrays with each other. Note how that passing an array to a function is the same as passing the address of the first element, i.e. the function subroutine has prepared a pointer to hold the address. Note that here is only one copy of the data in the arrays (not two) that is shared by the calling function (main()) and the called function (exch()).

```
1 /*cfunc.c: arrays w/functions*/
2  /*2010, Andy Chung */
3 void exch(int m[],int n[]);
4 int main(void){
5     int a[M]={31,11};
6     int b[M]={24,26};
7     exch(a,b);
8     return(0);
9 }
10 void exch(int m[],int n[]){
11     int k,tmp=0;
12     for(k=0;k<M;k++){
13         tmp=m[k];
14         m[k]=n[k];
15         n[k]=tmp;
16     }
17     return;
28 }
```



a. How many copies of the array are there?

As can be seen in the VP diagram, passing the array to the function creates a new pointer that points to the first element of the array. The elements of the array are not duplicated and so the function and the main program work with the same array elements.

6.9 Application

a. Dropping Object Problem

b. Optical Ray Tracing

c. Root Finding

6.10 Summary

Arrays allow for the linking of like data of the same type.

7. Appendix

7.1 ASCII Standard Encoding Table

Glyph	Dec	Hex	Glyph	Dec	Hex	Glyph	Dec	Hex
	32	20	@	64	40	`	96	60
!	33	21	A	65	41	a	97	61
"	34	22	B	66	42	b	98	62
#	35	23	C	67	43	c	99	63
$	36	24	D	68	44	d	100	64
%	37	25	E	69	45	e	101	65
&	38	26	F	70	46	f	102	66
'	39	27	G	71	47	g	103	67
(	40	28	H	72	48	h	104	68
)	41	29	I	73	49	i	105	69
*	42	2A	J	74	4A	j	106	6A
+	43	2B	K	75	4B	k	107	6B
,	44	2C	L	76	4C	l	108	6C
-	45	2D	M	77	4D	m	109	6D
.	46	2E	N	78	4E	n	110	6E
/	47	2F	O	79	4F	o	111	6F
0	48	30	P	80	50	p	112	70
1	49	31	Q	81	51	q	113	71
2	50	32	R	82	52	r	114	72
3	51	33	S	83	53	s	115	73
4	52	34	T	84	54	t	116	74
5	53	35	U	85	55	u	117	75
6	54	36	V	86	56	v	118	76
7	55	37	W	87	57	w	119	77
8	56	38	X	88	58	x	120	78
9	57	39	Y	89	59	y	121	79
:	58	3A	Z	90	5A	z	122	7A
;	59	3B	[	91	5B	{	123	7B
<	60	3C	\	92	5C	|	124	7C
=	61	3D	]	93	5D	}	125	7D
>	62	3E	^	94	5E	~	126	7E
?	63	3F	_	95	5F			

7.2 EO508 Coding Alchemy: Structure and Algorithms For Simulation 電腦模擬設計與實

EO508 is the followup course to EO109. In this course, we will first review the basices of ANSI-C, that is variables, pointers, arrays and records. After that we will go on to study programming in depth as we work to convert out UML diagrams into ANSI-C code. While the key goal of EO109 is *to learn* ANSI-C, the key goal of the followup course is *to use* ANSI-C to efficiently solve problems with code that others can understand.

Fig. D1 UML diagram of the key milestones and content of EO508 Computer Programming

7.3 Group Member List

1. Up to 4 per group. Select one member as the group leader. He will be responsible for the work of the group
2. Work together on tutorials
3. One member may be called at random to represent the group. The group's mark depends on his/her performance
4. Leader receives bonus marks if group does well.

Table E1: Group Members

	Group Name:			Group Number:		
	Role	**名字**	**Name**	**Student ID**	**Email**	**Hand-Phone**
1	**Leader*: 領導**					
2	**Member**					
3	**Member**					
4	**Member**					

Table E2: Group Member Progress Form

	Name		Milestones						Participation	
	English	**中文**	**1**	**2**	**3**	**4**	**5**	**6**	**c1**	**c2**
1										
2										
3										
4										

Table E3: Attendance Record After Semester Midpoint

#	Class Number [base ten (base twelve)]											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10(A)**	**11(B)**	**12(10)**
1												
2												
3												
4												

Table E4: Attendance Record After Semester Midpoint

#	Class Number[base ten (base twelve)]											
	13(11)	**14(12)**	**15(13)**	**16(14)**	**17(15)**	**18(16)**	**19(17)**	**20(18)**	**21(19)**	**22(1A)**	**23(1B)**	**24(20)**
1												
2												
3												
4												

7.4 Example Tests for Milestones

7.5 Tutorials

a. General

b. Falling Object

c. Optical Ray Tracing

d. Root Finding