Intensive Revision:

Basic Principles of Computer Programming in C

David Vernon

www. vernon.eu

The Computer Model

Information Processing

- When we process information, we do one of two things:
  - we change its representation (typically to enhance it)
  - we analyze it to extract its meaning
- How do we process information?

The Computer Model

Computer Software

- A computer program is a sequence of instructions (statements)
- Expressed in a given language (e.g. C)
- The language has a 'vocabulary'
  - a set of words
- The language has a 'grammar'
  - a set of rules about how words can be linked together
  - This is called the syntax of the language

A C Program

```
/* Example 1                                              */
/* This is a C program to ask you to type a letter */
/* and then to tell you what you typed                    */

#include <stdio.h>
main() {
   char letter;

   printf("Please type a letter & then press Return >>");
   scanf("%c",&letter);
   printf("You typed the letter %c", letter);
}
```

A C Program

```
/* Example 1                                                  */
/* This is a C program to ask you to type a letter */
/* and then to tell you what you typed                        */
```

- These are called comments
- They don't do anything, in that they don't cause the computer to process information
- Their purpose is simple to annotated the program: to tell someone reading the program what the program does

A C Program

```
main() {
```

- **`main`** is the name of the program; all C programs must have a **`main`** part.
- The open bracket (and close bracket) with nothing in between says that the program main doesn't work directly on information
  - we'll see later that we can put something in between the brackets and the program can use this 'something', typically to tell it what to do or to tell it what to process.

A C Program

```
main() {
```

- Finally, there is the { character
  - We call this an opening brace or curly bracket
  - This simply says that everything that follows until we meet the } character is part of the main program
  - The } is at the end of the program and we call it a closing brace or curly bracket

A C Program

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

- says ‘use the standard input/output utilities’ if you want to read and write messages
- There is a more technical interpretation of #include but we don’t need to bother with this for now

A C Program

```
char letter;
```

- causes the creation of a variable which can represent characters
- The name of the variable is letter and it can be used to represent the letter a or b or c or j or v or z
- Think of it as a general purpose letter holder (for one letter only)

A C Program

```
printf("Please type a letter & then press Return >>");
```

- This simply says print everything enclosed in the double quotes to the screen

A C Program

```
scanf("%c",&letter);
```

- is, in a way, the opposite of the printf line
- Instead of printing things, it reads things
- In this case, we are reading in a character
  - that's what the "`%c`" denotes
  - there are other symbols if we are reading other things such as numbers

A C Program

```
scanf("%c",&letter);
```

- Where do we put that character?
- We put it in the variable **`letter`**
- The symbol **`&`** is used to tell computer that we are allowing the value of letter to be changed, *i.e.* to be given (assigned) the value we read from the keyboard.

A C Program

```
scanf("%c",&letter);
```

- If we didn't have the **&** symbol, then we wouldn't be able to change the value of letter to the character we read.

A C Program

```
printf("You typed the letter %c", letter);
```

- What's different here is
  - the inclusion of the %c at the end of the message and
  - the addition of the name of our character variable, separated by a comma
  - The %c denotes a character and says 'we want to write out a character'
    - "What character?"
    - The character which is held in the variable letter

A C Program

- Each line of this program above is known as a statement
- Did you spot the semi-colons, *i.e.* the `;` characters, in all of the above?
  - In the C programming language, the semi-colons are used to temininate or finish-off each distinct statement

Summary

`/* This is a comment */`

`main()` is the name of our program

`#include <stdio.h>` means use the standard input and output factilities defined in the file **`stdio.h`**

Summary

`char` defines a character variable

`printf()` prints a message to the screen

`scanf()` reads something from the keyboard

`&` says you can change the value of what follows

`%c` says this is a character format

`;` terminates a statement

Summary

- The program does something like this:

```
Ask the user to type in a character
Read the character
Print out a message and the character we just read
```

- This is called pseudo-code
  - it isn't a real piece of computer code
  - but it tells us in computer-like way what the computer code would do.

Summary

- Later on we'll use pseudo-code to describe what a progam does before we actually write the computer program
- This 'preliminary draft' of the program is called design
- It is a crucial part of the overall process of writing computer software

The Computer Model

- Assuming we have designed and written a C program, we now we want to run it
- Let's do that together

A Second Example Program

In this section we will develop another slightly more complicated program

A Second Example

- Let's say we want the program to compare two numbers and tell us if they are the same
  - a simple task but all decisions basically come down to comparing two values
- This time, rather than start with the finished program, we'll take one step back and figure out for ourselves what it should look like
- We'll do this using the pseudo-code we met above

A Second Example

```
Ask the user to type in the first number
Read the first number
Ask the user to type in the second number
Read the second number
Compare the two number and print out a number
```

- This is fine but the last line is a bit too general
- To formulate exactly what we mean, let's try an example

A Second Example

- Let's say the first number is 10 and the second number is 12
- How do we compare them?
  - see if they are the same number.
  - How can we say that?
  - We could ask are they equal.
  - In C the way we check to see if things are equal is to say something like:

```
if  one_thing == another_thing
```

A Second Example

- In the number system, 10 comes before 12
- How might we formulate that? We simply say:

```
if  one_thing < another_thing
```

- Similarly, we might ask:

```
if  one_thing > another_thing
```

A Second Example

- That's progress but we need more
- What are we going to do if one thing is equal to another (or if one thing is less than another)
- Again, it's fairly straightforward. You just say

```
then do something
```

A Second Example

- And what if the answer to our question wasn't correct (or true)?
  - We have two options:
    - do nothing

      or
    - do something else

```
if  one_thing < another_thing
then do one thing
else do a different thing
```

A Second Example

- We normally write this as follows:

```
if  one_thing < another_thing
then
    do one thing
else
    do a different thing
```

- This indentation doesn't matter to the computer but it's very important readability by humans, especially as the programs become large

A Second Example

- This is called an if-then-else construct
- It can be stated more formally:

```
"if the following condition is TRUE then do one thing else
    (i.e. otherwise) do  a different thing"
```

- Note that if we didn't want to do anything at all if the test was not TRUE (that is, if it was FALSE) we'd just leave out the else part (often called the else-clause)

A Second Example

Now we can expand the 'Compare' statement:

```
ask the user to type in the first number
read the first number
ask the user to type in the second number
read the second number
if the first number == the second number
then
   print: the numbers are identical
else
   print: the numbers are different
```

```
/* Example 2                                                    */
/* This is a C program to ask you to enter two                  */
/* numbers; it then compares them and prints a                  */
/* message to say whether they are equal or not                 */

#include <stdio.h>

main() {

    int first_number, second_number;

    printf("Type a number and then press Enter >>");
    scanf("%d",&first_number);

    printf("Type another number and then press Enter >>");
    scanf("%d",&second_number);
```

```
/* Example 2                                              */
/* This is a C program to ask you to enter two            */
/* numbers; it then compares them and prints a            */
/* message to say whether they are equal or not           */

#include <stdio.h>

void main() {

   …TEXT DELETED…

   if (first_number == second_number)
      printf("The two numbers %d are identical",
              first_number);
   else
      printf("The two numbers %d and %d are different",
              first_number, second_number);
}
```

A Second Example

Several things to note:

- We now have two integer variables (**`first_number`**, **`second_number`**)
- We declared them in the same way as before but we separated them with a comma.

```
int first_number, second_number;
```

A Second Example

- We could also have said

```
int first_number;
int second_number;
```

- Note that we put an underscore instead of a space.
  As a general rule, you can't have spaces in the middle of the name of a variable.

A Second Example

- We changed the names of the variables in the **`scanf`** statements to **`first_number`** and **`second_number`**

- Note that we put a pair of brackets around the test in the **`if`** statement; these brackets MUST be there

```
if (first_number == second_number)
```

A Second Example

- In the second **`printf`** statement, we now have two variables: **`first_number`** and **`second_number`**

```
printf("The two numbers %d and %d are different",
        first_number, second_number);
```

  - These are separated by a comma
  - Because we have a second variable, the value of which is to be printed out in the message, we also need a second %d
  - Note that the value of the number will go into the message exactly where each %d is placed

A Second Example

- Finally, did you notice that we left out the word then in the C program?
- In C, the **`then`** is omitted
- Since it is normally required in many other programming languages and since it sounds more natural anyway, we'll keep on using it in our pseudo-code and then simply drop it when we write the corresponding C program
- Now, let's enter and run the program

Example No. 3

- Learn a little more C
- Begin to learn how to solve problems
  - software development is more about solving problems that it is about writing code
  - As we become more proficient at software development, we begin to take the underlying skills (or writing code) for granted

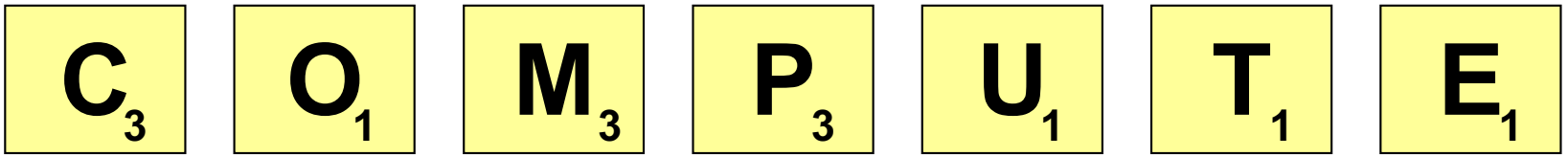
Example No. 3

S_1

- Compute the 'Scrabble' value of a collection of 7 letters

Scrabble

- a word-game
- players are awarded points for creating words from a random sample of seven letters
- Each letter carries a different point value
- the value of the word is the sum of the point values of each letter in the word
- We will assume we have a word with 7 letters … later we will modify the program to deal with words with between 1 and 7 letters.


C 3
O 1
M 3
P 3
U 1
T 1
E 1

Example No. 3

- First: how would you solve the problem if you had seven letters in front of you?
  - Probably, you'd pick up the first letter and find out its value
  - Then you'd pick up the second letter,
    - find out its value,
    - and add that to the first
  - Then you'd pick up the third,
    - find its value,
    - and add it to … what? To the running total
  - And again with the fourth, the fifth, the sixth, and the seventh

Example No. 3

- That's the beginning of a solution
  - we call this type of solution an algorithm
  - a step-by-step set of guidelines of what we have to do to get the answer we seek
- Let's try to write it out in pseudo-code

Example No. 3

```
Pick up the first letter
Find its value
Add this value to a running total

Pick up the second letter
Find its value
Add this to a running total
...
Pick up the seventh letter
Find its value
Add this to the running total

Print out the value of the running total
```

Example No. 3

What if we had not just 7 letters but 70

- The approach above is not going to work simply because we'd need to write out the above program segment 10 times
- Instead of writing out each and every step, we can say

  'do the following again and again and again until you have done it enough.'
- This means we loop around the same code again and again until we are finished

Example No. 3

In pseudo-code:

```
Do the following seven times
        Pick up a letter
        Find its value
        Add this to the running total
```

- We have reduced 21 lines of pseudo-code to 6
- If we were had 70 letters instead of 7: our savings would have been even greater (6 lines instead of 210)

Example No. 3

Note the way we indented the three statements with respect to the 'Do the following seven times'

- this indentation is a way of visually saying that these are the statements which are governed by the loop i.e. which are done seven times
- We'll see in a moment that the C language needs a little extra help in addition to the indentation but we'll retain the indentation nonetheless

Example No. 3

How do we know when we have looped seven times?

- We count! And when we have counted up to the total of seven, we're finished
- Of course, that means we start counting at 1
- Note that there are other way of counting
  - 11 to 17
  - 0 to 6

Example No. 3

The C programming language has shorthand way of saying all this

- It uses a counter
- gives it an initial value
- tells it to add a number each time round the loop
- and also says when to stop counting
- Actually, it doesn't say when to stop, it says when it is allowed to keep on counting – the exact opposite of when to stop
- That simply means: keep counting if we haven't reached the required number yet. Here's the C code:

Example No. 3

```
for (i=1; i<= 7; i = i+1)
```

- The **i=1** part set the value of **i** to **1**
- The **i<=7** says keep counting as long as **i** is less than or equal to seven
- The **i=i+1** simply adds **1** to **i** each time around the loop.
  - This means we'll count 1, 2, 3, 4, 5, 6, 7 and when we get to 8 we stop

Example No. 3

- What about the next pseudo-code statement: **`pick up a letter`**?

- We've done that already in our first program. We simply ask the user of the program for a letter and then we read it:

```
printf("Please type a letter & then press Return >>");
scanf("%c",&letter);
```

Example No. 3

Now the next part: *Find the value of the letter*

- Each letter has a given fixed value
- We encode these values in the program
- There are many ways of doing this; some are very efficient and elegant - others are more obvious and not so efficient
- We'll use the if-then-else approach
  - check to see if it is an A; if it is we set a variable to the value of the A letter
  - If it's not an A then we need to check B; if it is a B then we set the variable to the B value
  - Otherwise (else) we set check C, and then D, and so on.
  - (Note: there are better ways of doing this … more later).

Example No. 3

```
if letter == 'A'
then
   value is 10;
else
   if letter == 'B'
   then
      value is 2;
   else
....
           if letter == 'Z'
           then
             value is 10;
```

Example No. 3

Note something very important:

- we wrote the actual letter, not on its own, but in between two inverted commas (e.g. **`'Z'`**)
- Why didn't we just write the letter on its own (e.g. **`Z`**)?
- When we wrote the word **`letter`** we were referring to a variable called **`letter`**, an object into which we could place a value when we needed to (and change the value later, if we needed to)
- Similarly, if we write **`A`** then this is just a variable with a very short name
- We want the value of the letter **`'A'`**, *i.e.* the C representation of the letter **`'A'`**

Example No. 3

To make this a little clearer, let's consider another example.

- A short while ago, we used a variable called **`i`** as a counter
- **`i`** took on the numerical values 1, 2, 3, 4, 5, 6, and 7 in turn
- In this case, **`i`** is a number variable and we give it number values

Example No. 3

Actually, there are two types of numbers we could choose:

- real numbers (i.e. numbers with a decimal place)
- whole numbers (i.e. numbers with no decimal place)
- In computer terminology, we refer to real numbers as floating point numbers (there is a technical reason for this to do with their computer representation but we don't have to bother with it at this point)
- In C, we call floating point numbers **`float`** for short
- We refer to the whole numbers as integers or **`int`** for short

Example No. 3

- So what about our letters?
- Letters are just a subset of things called characters which also include exclamation mark (!), commas (,), full stops (.), and so on
- In fact anything you can type on a keyboard is a character
- In C, we call them **`char`** for short

Example No. 3

- Whenever we declare a variable in a program, we must say what type of variable it is
  - Is it an **`int`** (integer or whole number)
  - a **`float`** (floating point number or real number)
  - or a **`char`** (character)?
- We'll see how to do this in a moment when we write out this C program

Example No. 3

- Note that **`float`**, **`int`**, and **`char`** are not the only types in C
- There are many other types and we'll even define our own types
- However, the **`float`**, **`int`**, and **`char`** types form the basic building blocks for all these more advanced types
- Almost everything we can think of can be described (*i.e.* represented) by a sequence of characters or a collection of numbers

Example No. 3

- In Scrabble, not all the letters have different values: some letters have the same value

- We can use this fact to help reduce the number of if-then-else statements by grouping the letters which have the same value together:

Example No. 3

Letter	Value
A E I L N O R S T U	1
D G	2
B C M P	3
F H V W Y	4
K	5
J X	8
Z Q	10

Example No. 3

Our series of checks now becomes:

```
if letter == ‘ ’or letter == ‘ ’
then
   value is ;
else
   if letter == ‘ ’
   then
      value is ;
   else
...
          if letter == ‘ ’
          then
            value is 10;
```

Example No. 3

- We are almost ready to write our program
- We have decided on the actions we need to take – the logic of the program – now all we need to do is to decide what variables we need to represent our information
- In this example, we have only three pieces of information:
  - The letter entered by the user
  - its Scrabble value,
  - the total Scrabble value
  - (the `enter` character)

Example No. 3

- Have we forgotten anything?
- Yes, we have! We forgot our loop counter `i`
- Let's give these four variables names and types.

Variable Name	Variable Type
`letter`	`char`
`enter`	`char`
`scrabble_value`	`int`
`total_scrabble_value`	`int`
`i`	`int`

```
/* Example 3                                              */
/* Compute the total value of 7 Scrabble letters         */
/* Input:  the user is prompted to enter each letter     */
/*         in turn                                        */
/* Output: the program prints the sum of the seven       */
/*         individual letter values                       */

#include <stdio.h>

void main() {

   char letter, enter;
   int scrabble_value,
       total_scrabble_value,
       i;

   /* initialize variables */

   total_scrabble_value = 0;

   /* use a for loop to read seven variables */
```

```
   /* use a for loop to read seven variables */

   for (i=0; i < 7; i++) {
      printf("Please type a letter and then press Return >>");
      scanf("%c",&letter);
      scanf("%c",&enter); /* skip enter character */

      if ((letter == ' ') || (letter == ' '))
         scrabble_value = 0;
      else
         if ((letter == ' ') || (letter == ' '))
            scrabble_value = 0;
                  MORE....

      /* now add the value to the total */

      total_scrabble_value = total_scrabble_value + scrabble_value;
   }

   printf("The Scrabble value of the seven letters is %d",
           total_scrabble_value);
}
```

Example No. 3

Again, several things are worth noting about this program:

- Did you notice the { which follows immediately after the for statement?
- In C, only one statement is part of the loop
  (*i.e.* only one statement is repeated again and again)
- This is the statement which follows immediately after the for statement
- If we want more than one statement to be repeated (and we certainly do in this case) then we simply enclose them in a pair of braces or curly brackets

Example No. 3

- We call the statement or group of statements to be repeated the body of the loop

- We indent the body of the loop with respect to the loop control statement to show the structure of the logic of the program visually

Example No. 3

Note how we translated our pseudo-code statement:

```
if letter == ‘A’or letter == ‘B’
```

into

```
if ((letter == ‘A’) ¦¦ (letter == ‘B’))
```

Example No. 3

We see three things:

- First, we use the two-character symbol **||** instead of the word or
  - Note that these two characters go side-by-side; we can't put any spaces in between them.

Example No. 3

We see three things:

- Second, we put brackets around each test
  - This isn't strictly necessary but it's good practice
  - C has its own rules about how it goes about figuring out whether or not the if statement is true or false and, on occasion, the rules can be tricky to follow
  - However, if we put brackets around the test like we have, it's clear how things work
  - We call one of these tests (*e.g.* letter == 'A') a relational expression because it finds out the relationship between the object on the left (in this case the variable letter) and the object on the right (the character value 'A')

Example No. 3

We see three things:

- Thirdly, we put a pair of braces around the entire if test just as we did in the second example.

Example No. 3

As a general rule, when we use variables we should never assume anything about their initial values

- Variables are like boxes for holding things
- When you declare a variable, you get a box but you have no idea what's in it
- So, we always – repeat, always – initialize the variables with some value before we ever use that variable
- Think about it: it makes a lot of sense to put something into the box before taking anything out!

Example No. 3

- When we declared the four variables, we put one on each line but we didn't have to
  - We do it just to make the presentation visually appealing and to improve readability
- We assigned a value to **`total_scrabble_value`** with the **=** operator
- A very common mistake made by people who are new to C is to get the two operators **=** and **==** mixed up

Example No. 3

The difference is crucial:

= is an assignment operator and assigns values to variables

== is a relational operator and it is used to test the equality of the two objects on either side of it. Actually, these two objects are called operands: the things that the operators operate on!)

Example No. 3

At this point, we've learned quite a lot of C programming.
We know how to:

- Write a complete program
- Read and write messages, numbers, characters
- Use an if [then] else statement
- Use a loop
- Know how to declare variables
- Know how to assign values to variables

Practise Solving Problems & Creating Algorithms

Key Skills

- ❑ Manual walkthroughs
- ❑ Creative thinking about the problem
- ❑ Spotting patterns
- ❑ Using pseudo-code

Practise Solving Problems & Creating Algorithms

Manual walkthroughs

- Do it
- Do it again and watch yourself doing it
- Write down what you saw
- Polish (refine) what you wrote into pseudo-code
  - Assignment
  - Iteration / loops
  - Conditional execution
- Manually execute your pseudo-code on input with a window (with a letter-box view, i.e. line by line)

Example 4:
Comparing Numbers

- Design and write a program to prompt the user three times and reads three numbers

- It should then compare these three numbers and tell the user whether
  - all three numbers are the same
  - all three numbers are different
  - just two numbers are the same
    - tell the user which two numbers they are

- The program should continues to ask the user for input until he enters three zeros

In pseudo-code:

```
n1 = 1; n2 = 1; n3 = 1;
While the three numbers are not all zero
      (i.e. NOT(all three numbers are zero))
do the following

     Read a number n1
     Read a number n2
     Read a number n3

     if n1 == n2 and n2 == n3 and n1 == n3
     then
          print the numbers are all the same
     else
        if n1 != n2 and n2 != n3 and n1 != n3
        then
            print the numbers are all different
```

```
else
   /* two are the same … which are they? */

    if n1 == n2
    then
       print two are the same:
       the first and the second: n1, n2
    else
       if n1 == n3
          print two are the same:
          the first and the third: n1, n3
      else
           print two are the same:
           the second and the third: n2, n3
```

```
/* A program to prompt the user three times and reads three numbers. */
/* It compare these three numbers and tell the user whether          */
/*                                                                   */
/* - all three numbers are the same                                  */
/* - all three numbers are different                                 */
/* - just two numbers are the same                                   */
/*   in this case, it also tells the user which two numbers they are */
/*                                                                   */
/* The program continues to ask the user for input until he enters   */
/* three zeros.                                                      */

#include <stdio.h>

#define TRUE  1
#define FALSE 0

void main() {
```

```
int n1, n2, n3;

/* give the numbers initial values */

n1 = 1;
n2 = 1;
n3 = 1;

/* while the three numbers are not all zero */

while (! ((n1 == 0) && (n2 == 0) && (n3 == 0))) {

   printf("Please enter the first number  >>");
   scanf("%d",&n1);

   printf("Please enter the second number >>");
   scanf("%d",&n2);

   printf("Please enter the third number  >>");
   scanf("%d",&n3);
```

```
/* check to see if they are all the same */

if ((n1 == n2) && (n2 == n3) && (n1 == n3)) {
   printf(" The three numbers are all the same. \n");
}
else {
   if ((n1 != n2) && (n2 != n3) && (n1 != n3)) {
       printf("The three numbers are all the different.\n");
    }
   else {

      /* two are the same -  which are they? */

      if (n1 == n2) {
          printf("The first and second numbers are the same: %d\n”,
        n1, n2);
       }
       else {
          if (n2 == n3) {
             printf("The second and third numbers are the same: %d\n",
           n2, n3);
          }
```

```
                else {  /* no need to check if the first and third    */
                        /* are the same ... it's the only possibility */
              printf("The first and third numbers are the same:%d\n",
                     n1, n2);
                }
             }
          }
       }
    }
}
```

Example No. 4

Note how we translated our pseudo-code statement:

```
if n1 == n2 and n2 == n3 and n1 == n3
```

into

```
if ((n1 == n2) && (n2 == n3) && (n1 == n3))
```

Example No. 4

- We used a `while` loop rather than a `for` loop
- It has the format:

```
while <condition is true>
    statement
```

- `statement` is executed as long as the condition is true
- Obviously, we need a compound statement `{ }`
  if we want to repeat several statements

Example No. 4

```
while (! ((n1 == 0) && (n2 == 0) && (n3 == 0))) {

    /* all the statements are repeatedly     */
    /* executed while this condition is true */
    /* i.e. while NOT all numbers are zero   */
    /* so we finish when they are all zero!  */

}
```

Example No. 4

- We used the `!` operator for the logical NOT operation

- We used the `&&` operator for the logical AND operation

Example No. 4

- We could also have used a third type of loop:
  the `do while` loop

- It has the format:

```
do
    statement
while <condition is true>
```

- `statement` is executed as long as the condition is true

Example No. 4

```
while (! ((n1 == 0) && (n2 == 0) && (n3 == 0))) {

    /* all the statements in repeatedly       */
    /* executed while this condition is true */
    /* i.e. while NOT all numbers are zero   */
    /* so we finish when they are all zero!  */
    /*                                       */
    /* we could have written this in another */
    /* way:                                  */
    /* while ((n1!=0) || (n2!=0) || (n3!=0)) */
}
```

A More Formal Treatment of C

Programs:
Statements + Data Constructs

- Every C program is built up from two different types of construct:
  - Data constructs
    - often called data-structures
    - these hold, or represent, information
  - Statements
    - these define actions
    - the actions usually process the data
      - input data
      - transform data
      - output data

Programs:
Statements + Data Constructs

- Each program contains one or more *functions*, one of which is called **`main`**

```
int main()
{          /* beginning of body */
           …
}          /* end of body       */
```

- A function is a logical collection of statements which can be referred to by the function name
  - e.g. cos(x) is a function which computes the cosine of an angle x

Data Constructs

Declarations

Define variables:

- name
  - a sequence of characters by which we refer to that variable: this is called an identifier
- type
  - a definition of all the possible values that the variable can have.
- For example:

  ```
  int counter;
  ```

  - declares a variable called `counter` which can take on any of the integer values
  - We say the counter variable is *instantiated*

Data Constructs

Declarations

- Optionally, declarations initialize variables
- For example:
  - `int counter = 0;`
  - declares a counter of type integer and gives it an initial value zero

Data Constructs

- An identifier is a sequence of characters in which only letters, digits, and underscore _ may occur

- C is case sensitive ... upper and lower case letters are different

  - `int counter = 0;`
  - `int Counter = 0; /* different */`

Data Constructs

- There are three basic types in C:

  `int`

  `float`

  `char`

  - `int` variables can have integer values
  - `float` variable can have real number (floating point) values
  - `char` variables can have character values

- So, how do we write integer, floating point, and character values in C?

Data Constructs

Integer values:

- 123 (decimal)
- 0777 (octal)
- 0xFF3A (hexadecimal)
- 123L (decimal, long - 4 bytes)
- 12U (decimal, unsigned - no sign bit)

Data Constructs

Floating Point Values

- Type **`float`**

  82.247

  .63l

- Type **`double`** (8 byte representation)

  82.247

  .63

  83.

  47e-4

  1.25E7

  61.e+4

Data Constructs

Character Values

- '**A**' enclosed in single quotes
- Special characters (escape sequences)
  - '**\n**' newline, go to the beginning of the next line
  - '**\r**' carriage return, back to the beginning the current line
  - '**\t**' horizontal tab
  - '**\v**' vertical tab
  - '**\b**' backspace
  - '**\f**' form feed
  - '**\a**' audible alert

Data Constructs

Character Values

'\\'	backslash
'\''	single quote
'\"'	double quote
'**?**'	question mark
'***OOO***'	octal number
'**x*hh***'	hex number

Data Constructs

Floating Values

Type	Number of Bytes
`float`	4
`double`	8
`long double`	10

Implementation dependent

Data Constructs

String Values

- String literal
- String

  `"How many numbers?"`

  `"a"`
- `"a"` is not the same as `'a'`
- A string is an array of characters terminated by the escape sequence `'\0'`
- Other escape sequences can be used in string literals, e.g. `"How many\nnumbers?"`

Statements: Comments

- `/* text of comment */`

- The characters contained in a comment are ignored by the compiler

Statements: Assignment

Assignment Statement

- Syntax:

  variable = expression ;

- For example

```
x = p + q * r;
```

- Operators:

  `+, *, =`

- Operands:

  `p, q, r, x`

Statements: Unary Operators

Unary operator: -, +

```
neg = - epsilon;
pos = + epsilon;
```

Statements: Binary Operators

Binary operators: `+, -, *, /, %`

```
a = b + c;
```

- Integer overflow is not detected
- Results of division depends on the types of the operands

```
float fa = 1.0, fb = 3.0;
int a = 1, b = 3;
```

what is the value of `fa/fb;`

what is the value of `a/b;`

Statements: Modulus

Remainder on integer division

```
%

39 % 5     /* value of this expression? */
```

Statements: Arithmetic Operations

Assignment and addition

```
x = x + a
x += a
```

- These are expressions and yield a value as well as performing an assignment
- We make them statements by adding ;

```
x = x + a;
x += a;
```

Statements: Arithmetic Operations

Other assignment operators

```
x -= a
x *= a
x /= a
x %= a

++i     /* increment operator: i += 1 */
--i     /* decrement operator: i -= 1 */
```

Statements: Arithmetic Operations

Other assignment operators

```
/* value of expression = new value of i */

++i     // increment operator: i += 1
--i     // decrement operator: i -= 1

/* value of expression = old value of i */

i++     // increment operator: i += 1
i--     // decrement operator: i -= 1
```

Types, Variables, and Assignments

Type	Number of Bytes
`char`	1
`short (short int)`	2
`int`	2
`long (long int)`	4
`float`	4
`double`	8
`long double`	10

Types, Variables, and Assignments

Use **`sizeof`** to find the size of a type

e.g.

```
sizeof (double)
```

Statements: output

For output, use (C library function) `printf`

```
char ch = 'A'; int i = 0;
float f = 1.1; double ff = 3.14159;

printf("ch = %c, i = %d\n", ch, i);
printf("f = %10f, ff = %20.15f\n", f, ff);
```

Statements: output

- To use `printf` you must include `stdio.h`

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

- syntax:

```
printf(<format string>,
       <list of variables>);
```

- `<format string>`

  String containing text to be printed and conversion specifications

Statements: output

- Conversion specifications

```
%c          characters
%d          decimals
%f          floats or doubles
%s          strings
```

- can also include field width specifications:

```
%m.kf       m is the field width
                  k is the number of digits
                  after the decimal point
```

Statements: input

- For input, use (C library function) `scanf`

```
char ch = 'A'; int i = 0;
float f = 1.1; double ff = 3.14159;

scanf("%c %d %f %lf", &ch, &i, &f, &ff);
```

- The ampersand `&` is essential
  - It takes the address of the variable that follows
  - scanf expects only variables

Compound Statement

- Statements describe actions
- Expressions yield values
- We use braces { } to build a complex, i.e. compound, statement from simpler ones
- Typically, we use compound statements in places where the syntax allows only one statement

```
{   x = a + b;
    y = a - b;
}
```

Comparison and Logical Operators

Operator	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to
&&	logical AND
||	logical OR
!	logical NOT

Comparison and Logical Operators

- `<, >, <=, >=` are relational operators
- `==` and `!=` are equality operators
  - relational operators have a higher precedence than equality operators (i.e. they are evaluated first in an expression)
  - Expression formed with these operators yield one of two possible values

    0 means false

    1 means true

    Both are of type int

Conditional Statements

- Syntax

```
if (expression)
    statement1
else
    statement2
```

  - The else clause is optional

- Semantics
  - `statement1` is executed if the value of expression is non-zero
  - `statement2` is executed if the value of expression is zero

Conditional Statements

- Where appropriate *statement1* and *statement2* can be compound statements

```
if (a >= b)
{   x = 0;
   if (a >= b+1)
   {   xx = 0;
       yy = -1;
   }
   else
   {   xx = 100;
       yy = 200;
   }
}
```

Iteration Statements

- while-statement syntax

```
while (expression)
    statement
```

- semantics
  - `statement` is executed (repeatedly) as long as `expression` is non-zero (true)
  - `expression` is evaluated before entry to the loop

Iteration Statements

```
/* compute s = 1 + 2 + ... + n */

s = 0;
i = 1;
while (i <= n)
{   s += i;
    i++;
}
```

Iteration Statements

- do-statement syntax

```
do
    statement
while (expression);
```

- semantics
  - `statement` is executed (repeatedly) as long as `expression` is non-zero (true)
  - `expression` is evaluated after entry to the loop

Iteration Statements

```
/* compute s = 1 + 2 + ... + n */

s = 0;
i = 1;
do                  /* incorrect if n == 0 */
{   s += i;
    i++;
} while (i <= n)
```

Iteration Statements

- for-statement

  `for (`*`statement1 expression2; expression3`*`)`
  `statement2`

- semantics
  - *`statement1`* is executed
  - *`statement2`* is executed (repeatedly) as long as *`expression2`* is true (non-zero)
  - *`expression3`* is executed after each iteration (i.e. after each execution of *`statement2`*)
  - *`expression2`* is evaluated before entry to the loop

Iteration Statements

```
/* compute s = 1 + 2 + ... + n */

s = 0;
for (i = 1; i <= n; i++)
   s += i;
```

Switch

- `switch (expression) statement`
  - the `switch` statement causes an immediate jump to the statement whose label matches the value of `expression`
  - `statement` is normally a compound statement with several statements and several labels
  - `expression` must be of type `int`, `char`

Switch

```
/* example of the switch statement */

switch (letter)
{  case ‘N’: printf( “New York\n” );
             break;
   case ‘L’: printf( “London\n” );
             break;
   case ‘A’: printf( “Amsterdam\n” );
             break;
   default: printf( “Somewhere else\n”);
             break;
}
```

Switch

```
/* example of the switch statement */

switch (letter)
{  case 'N': case 'n': printf( "New York\n");
                       break;
   case 'L': case 'l': printf( "London\n");
                       break;
   case 'A': case 'a': printf( "Amsterdam\n");
                       break;
   default: printf( "Somewhere else\n");
            break;
}
```

Bit Manipulation

The following bit manipulation operators can be applied to integer operands:

&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Inversion of all bits
<<	Shift left
>>	Shift right

File Input/Output

- So far, we have read all input from the keyboard and written all output to the screen
- Often, we want to be able to read and write information to and from files on disk
- We can use everything we learned about **`printf`** and **`scanf`** to do this, with just a few changes

File Input

- Declare a file pointer

```
FILE *fp;
```

- Open the file for reading

```
fp=fopen(“input.txt”, "r");
```

- Read from the file

```
fscanf(fp, “%d”, &number); /* assume int number */
```

- Close the file

```
fclose(fp);
```

File Input

We may also need to check if the file open operation was successful

```
fp=fopen(“input.txt”, "r");
if (fp == 0) {
      printf("unable to open input.txt\n");
      exit();
   }
```

File Input

- We can also check if we have reached the end of the file

```
int end_of_file;

end_of_file = fscanf(fp,”%d”,&number);

if (end_of_file == EOF) {
   printf(”end of input.txt reached\n");
   exit();
}
```

- Usually use this to control a loop

File Input

```
int end_of_file;

end_of_file = fscanf(fp_in, "%d", &n);
while (end_of_file != EOF)
{
   /* do something with n here */

   ……

   /* now read another value */
   end_of_file = fscanf(fp_in, "%d", &n);
}
```

```
FILE *fp;                  /* declare a file pointer   */
int  number;               /* number to be read        */

/* open the file: filename is input.txt    */
/*                r means open for reading */

fp=fopen(“input.txt”, "r");

/* check to see if we could open the file  */
/* if not, issue an error message and quit */

if (fp == 0) {
   printf("unable to open input.txt\n");
   exit();
}
```

```
/* read a number */

fscanf(fp, "%d", &number);    /* similar to scanf  */

/* rest of processing goes here … */


/*  when you have finished reading all the numbers, */
/*  close the file                                  */

fclose(fp);
```

File Output

- Declare a file pointer

```
FILE *fp;
```

- Open the file for writing

```
fp=fopen("output.txt", "w");
```

- Write to the file

```
fprintf(fp, "%d", number); /* assume int number */
```

- Close the file

```
fclose(fp);
```

```
FILE *fp_out;              /* declare a file pointer   */
float pi=3.14;             /* number to be written     */

/* open the file: filename is output.txt    */
/*                w means open for reading  */

fp_out=fopen("output.txt", "w");

/* check to see if we could open the file  */
/* if not, issue an error message and quit */

if (fp_out == 0) {
   printf("unable to open output.txt\n");
   exit();
}
fprintf(fp_out, "%f%, pi);
fclose(fp_out);
```

File Input/Output

Exercise

- Write a program to read a file containing five integers and find the maximum value
- Input: in.txt
- Output: maximum is written to the screen

File Input/Output

Exercise

- Write a program to copy a file of integers to another file
- Input: in.txt
- Output: out.txt
- Note: we don't know how many numbers are in the file

File Input/Output

Exercise

- Write a program to read a file of floating point numbers and find the maximum value
- Input: in.txt
- Output: max.txt
- Note: we don't know how many numbers are in the input file

Functions

C allows us to create our own functions

- We normally use functions if we have a well-defined segment of code which we wish to use many times in a program
- We can then use that segment of code simply by referring to the function name

Functions

```
Function A
```

```
Function B
```

```
Main(){
    A();
    B();
}
```

Question:
How do we provide
the functions with information
or data to use?

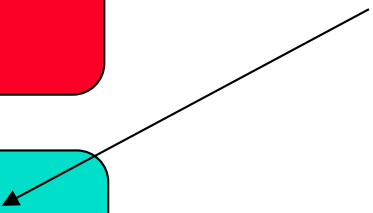
Answer:
We use parameters
and arguments

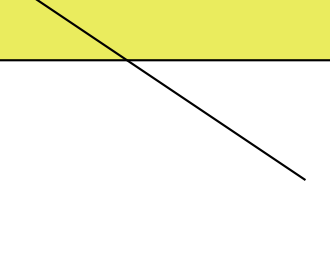
Functions

```
void A(int x)
```

```
void B(float y)
```

```
Main(){
    float z;
    A(10);
    B(z);
}
```





The value of z is passed to the function B

This is called parameter passing

Functions

- Function Definition:
  - Function name
  - Function type
  - Function parameters
  - Constituent statements
  - Local variables
- Function Call
  - Passing parameters (arguments)
  - Execution of a function

```
/* Definition of a function to print a message */

void message(int message_number)
{
   switch (message_number)
   {
      case 1:  printf("Hello World");
               break;
      case 2:  printf("Goodbye World");
               break;
      default: printf("unknown message number");
               break;
   }
}
```

```
/* Example function call */

void main()
{
   int x=2;

   message(1);  /* what will happen? */
   message(x);
   message(0);

}
```

Functions

Points to notice:

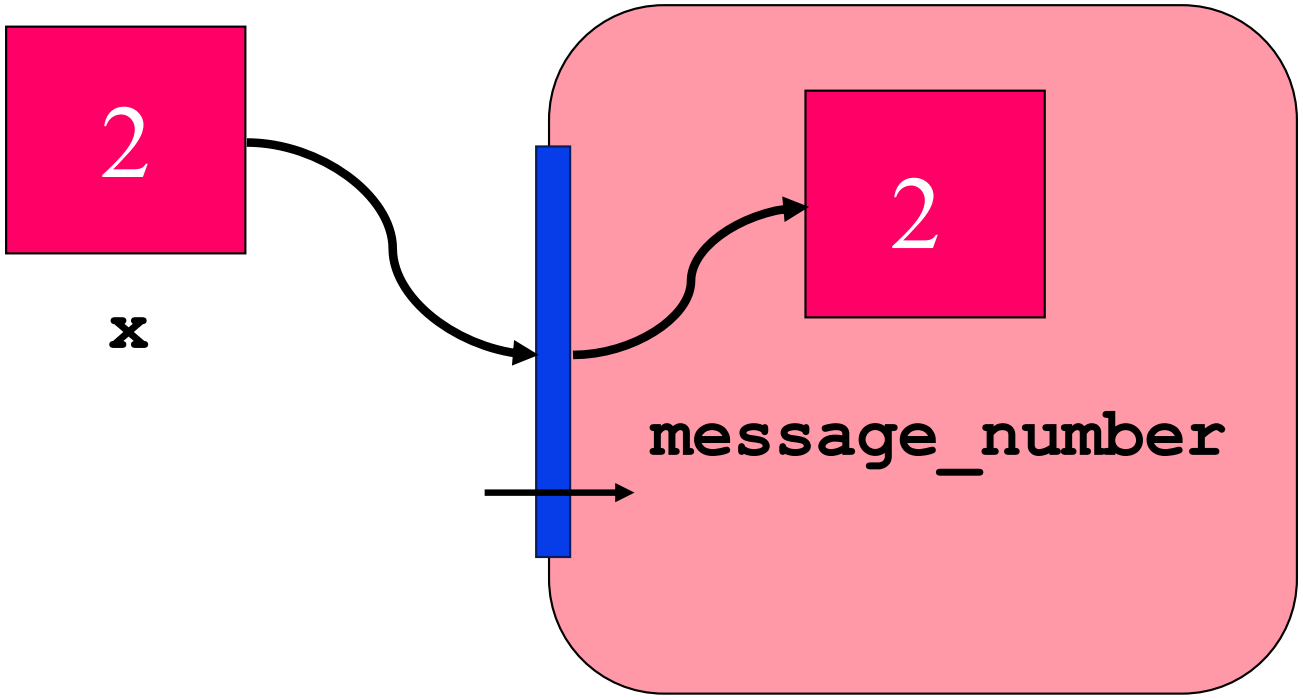
- the function name is `message` and we call the function simply by naming it
- the function has a type `void` (more later)
- the function has one parameter: `message_number`
- when we call the function we must provide a corresponding argument (and the types of the parameter and argument must be the same).

Functions

Points to notice:

- the argument doesn't have to be a variable in this case; we can use a literal value
- this is true only because we are passing the parameter by value
- this means we take a copy of the argument and pass to the function
- the function is then free to do whatever it likes with it and it will have no effect on the argument

Functions



Pass parameter by value:

make a copy of the argument

`message`

Functions

- What happens if we DO want to change the argument?
- Pass the parameter by reference
- This means we don't make a copy but use the **address** of the argument instead

Functions

```
/* Example function call */

void main()
{
   int x=2, check;

   message2(1, &check);
   message2(x, &check);
   message2(0, &check);

}
```

```
/* Pass a parameter by reference                    */

void message2(int message_number, int *valid)
{
   switch (message_number)
   {
      case 1:  printf("Hello World");
               *valid = 1;
               break;
      case 2:  printf("Goodbye World");
               *valid = 1;
               break;
      default: printf("unknown message");
               *valid =0;
               break;
   }
}
```

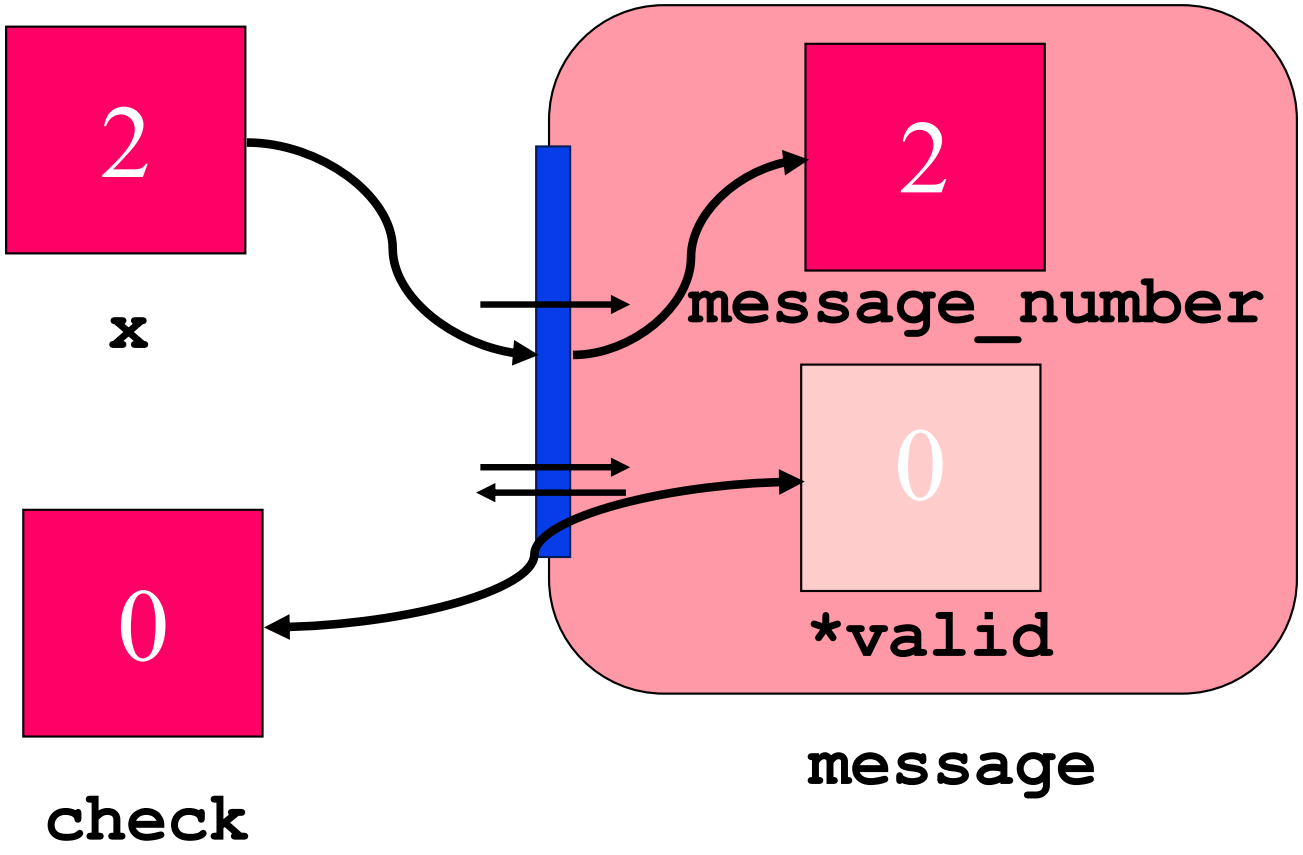
Functions

```
/* Example function call */

void main()
{
   int x=2, check;

   message2(1, &check);  /* what will happen? */
   message2(x, &check);
   message2(0, &check);

}
```

Functions

- Points to notice:
  - we now have two parameters and two arguments; both agree in type
  - The first parameter is passed by value
  - The second parameter is passed by reference
  - Anything we do to the parameter passed by reference also happens to the corresponding argument **because they both refer to the same variable**

Functions



Pass parameter by reference

Functions
Altering Variables via Parameters

C++ allows reference parameters

```
void swap1(int &x, int &y)
{   int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Functions
Altering Variables via Parameters

- C does not allow reference parameters

- Instead of passing parameters by reference
  - we pass the address of the argument
  - and access the parameter indirectly in the function

- `&` (unary operator)

  address of the object given by the operand

- `*` (unary operator)

  object that has the address given by the operand

Functions
Altering Variables via Parameters

```
void swap(int *p, int *q)
{   int temp;
    temp = *p;
    *p = *q;
    *q = temp;
}

swap(&i, &j); // function call
```

Functions
Altering Variables via Parameters

Pointers

- `*p` has type `int`
- But `p` is the parameter, not `*p`
- Variables that have addresses as their values are called pointers
- `p` is a pointer
- The type of `p` is pointer-to-int

Functions

- There is one other way of getting a result out of a function: through the function name
- The function has a type so we can assign a value to it
- We do this using the **`return`** statement
- We can then use the function value in the calling program by treating the function just like an ordinary variable

```
/* definition of a function which returns a value   */

int message3(int message_number)
{  int valid; /* declare a local variable */
   switch (message_number)
   {
      case 1:  printf("Hello World");
               valid = 1;
               break;
      case 2:  printf("Goodbye World");
               valid = 1;
               break;
      default: printf("unknown message");
               valid = 0;
               break;
   }
   return(valid);
}
```

Functions

```
/* Example function call */

void main()
{
   int x=2, check;

   check = message3(1);  /* what will happen? */
   check = message3(x);
   check = message3(0);
   if (message3(x) == 0)
      exit();
}
```

Functions

Points to notice:

- the type of the variable (or value) used in the return statement must be the same as the type of the function
- we declared a local variable called **`valid`**;
  this variable is only defined inside the function and is not visible (*i.e.* usable) outside it
- it disappears when we leave the function

Functions

Example 1:

- write a function to return the maximum of two floating point numbers, each passed as a parameter

Functions

Example 2:

- Write a function to sort two floating point numbers, each passed as a parameter
- On returning from the function, the first argument should be the smaller of the two number
- The function should return 0 if the numbers are different, 1 if they are the same

Data Constructs - Arrays

- We know that there are three basic types in C:

```
int
float
Char
```

- However, C also allows us to organize these basic types into more structured types

- One of these is the array

Data Constructs - Arrays

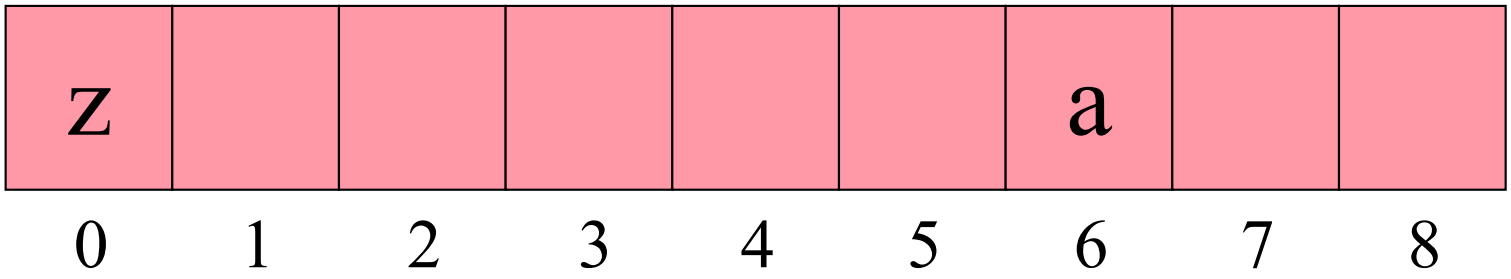
- An array is a collection of many variables which are all of the same type

- We declare an array by

  - giving its base type (i.e. the type we are collecting together)
  - giving the number of elements in the collection
  - ```
    int a[30]; /* declare an array of */
               /* 30 integers          */
    ```
    ```

Data Constructs - Arrays

- We access (i.e. use) an individual element by
  - writing the array name
  - and the element number in square brackets
  - ```
    printf("%d", a[6]); /* print out the   */
                        /* value of the    */
                        /* SEVENTH element!*/
    ```
- In C, the first element is element 0
    ```

Data Constructs - Arrays

In C, the first element is element 0




```
char example[9]; /* an array of 9 characters */
example[0] = 'z';
example[6] = 'a';
```

Data Constructs - Arrays

- We can have any type as the base type (int, char, float, even other structured types)

- You must say how many elements are in an array when you declare it

Data Constructs - Arrays

Example 1:

- read a sequence of 25 integer numbers from a file called `input.txt`
- compute the average of these numbers, but ignore any numbers with the value zero
- replace all occurrences of the number zero with the average
- write out the 5 numbers to a file called `output.txt`

Data Constructs - Arrays

Example 2:

- Do each of the four main tasks using a function
  - input_numbers
  - compute_average
  - replace_number
  - output_numbers

- Point to watch: how do we pass an array as a parameter?

Data Constructs - Arrays

Passing arrays as arguments to a function

- In C, you can't pass arrays by value
- they are only passed by reference
- However, to do this **you DON'T use the & operator** as you would with a normal variable
- The argument is simply the array name
- The parameter is declared as follows:

Data Constructs - Arrays

```
void function_x(int array_1[])
{ …
}
```

- This declares a parameter called **`array_1`** which is of type array of integer
- You don't have to say how big the array is (but see an exception later for 2-D arrays)

```
#include <stdio.h>

/* function to initialize an array of integers to  */
/* a given initial value                           */
/* parameters passed:  a, the array of integers    */
/*                     n, number of elements to be */
/*                        initialized              */
/*                     val, the initial value      */

void initialize(int a[], int n, int val)
{
   int i;
   for (i=0; i<n; i++) {
      a[i] = val;
}

void main()
{
   int x[40];
   initialize(x, 40, 0); /* init array values to zero */
}
```

Data Constructs - Arrays

2-D arrays

- you can declare a 2-D array in C as follows:

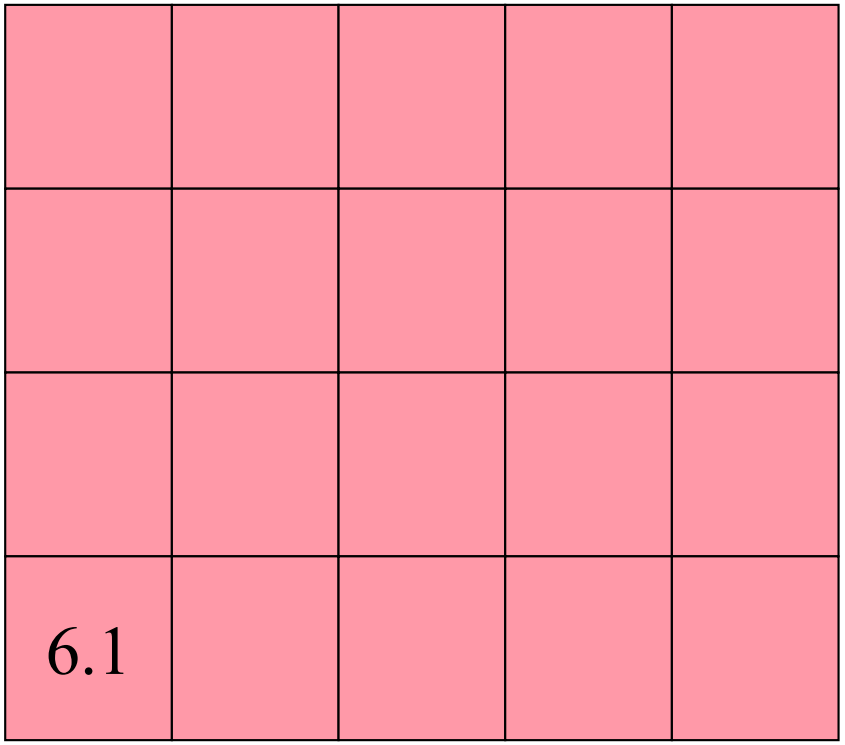
```
float array[4][5];
```

- This creates a 2-D array of floating point numbers with 4 rows and 5 columns

Data Constructs - Arrays

2-D Arrays

```
float a[4][5];
a[3][0] = 6.1;
```



Data Constructs - Arrays

Passing 2-D arrays as arguments to a function

- The argument is simply the array name
- However, the parameter is declared as follows:

```
void function_y(float array_1[][5])
{  ...
}
```

You MUST provide the number of columns

```
#include <stdio.h>

/* function to initialize a 2-D array of floats to      *,
/* a given initial value                                *,
/* parameters passed:  b,    the array of floats        *,
/*                     x, y, dimensions to be           *,
/*                           initialized                *,
/*                     val,  the initial value          *,

void init_2D(float b[][6], int x, int y, float val)
{
   int i, j;
   for (i=0; i<x; i++) {
      for (j=0; j<y; j++) {
         b[i][j] = val;
      }
   }
}
```

```
void main()
{
   float x[5][6];

   /* initialize the entire array to zero */

   init_2D(x, 5, 6, 0);


   /* initialize the first row to 7 */

   init_2D(x, 1, 6, 7);
}
```

Data Constructs - Arrays

character arrays

- a 1-D array of characters is used to represent a string

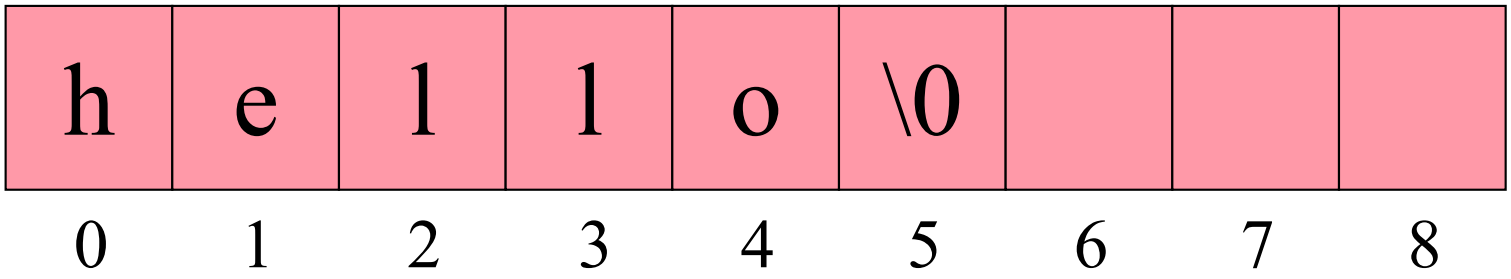
```
char s[12];
```

- a string value is written "`hello world`"

- all strings have an extra character at the end to mark the end of the sequence of characters: `'\0'`

Data Constructs - Arrays

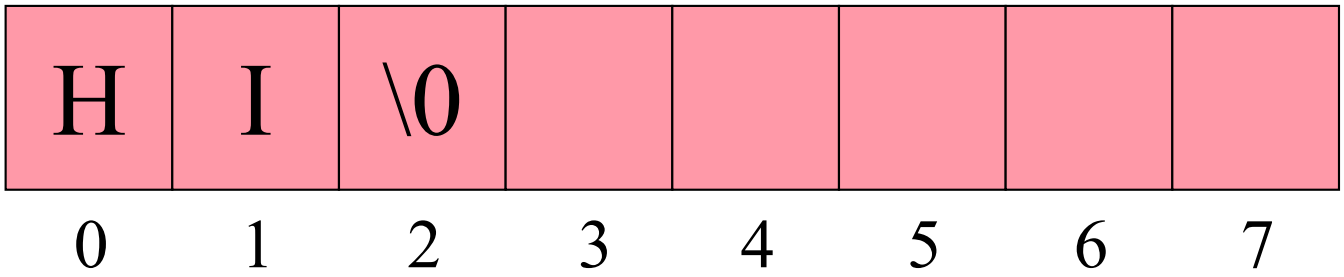
Strings: character arrays




```
char example[9]; /* an array of 9 characters */
example = “hello”;
printf(“%s”, example);
```

Data Constructs - Arrays

Strings: character arrays




```
char example2[8]; /* an array of 8 characters *,
example2[0] = 'H';
example2[1] = 'I';
example2[2] = '\0';
printf("%s", example2);
```

Data Constructs - Arrays

character arrays

- the C library contains many useful string handling functions

```
#include <string.h>
```

- More on this in the next unit

Data Constructs - Structures

- Arrays allowed us to group together collections of variables of the same type
- Structures allow us to group together collections of variables of different types
- The members of a structure can be simple types (char, float, int)
- But they can also be structured types (arrays, structures)

Data Constructs - Structures

- We declare a structure using the keyword `struct` and enclosing a list of members in braces, e.g.

```
struct colour {
    int r;
    int g;
    int b;
 }
```

- Note: this declares a new type, not a variable.

Data Constructs - Structures

- To create a variable of type colour, we write:

```
struct colour white, black;
```

- We access the members by writing the name of the structure variable followed by a dot and the name of the member:

```
white.r = 255;
white.g = 255;
white.b = 255;
black.r = 0; black.g = 0; black.b = 0;
```

Data Constructs - Structures

For example, to print the r, g, b colour values of the variable white we would write

```
printf("%d,%d,%d", white.r, white.g, white.b);
```

Data Constructs - Structures

- Structures can be nested (i.e. a structure can be a member of another structures)

- For example:

```
struct point {
    int x;
    int y;
    struct colour clr;
}
```

Data Constructs - Structures

And we can access each member as follows:

```
struct point p1, p2;
p1.x = 10;
p1.y = 20;
p1.clr.r = 128;
p1.clr.g = 128;
p1.clr.b = 128;

p2 = p1;  /* structure assignment */
```

Data Constructs - Structures

We can also have arrays of structures

```
struct point p[10];
p[3].x = 10;
p[3].y = 20;
p[3].clr.r = 128;
p[3].clr.g = 128;
p[3].clr.b = 128;

p[0] = p[3];  /* structure assignment */
```

Data Constructs - Structures

Exercise: write a program to

- read a file of integer coordinates and RGB values
- store the input in an array of points
- write out the colour of each point to an output file
- the input file is points.txt
- the output file is colours.txt

Data Constructs - Structures

- Example input:

  1 2 23 45 67
  2 3 44 55 0
  3 3 0 0 0
  4 4 255 255 255

- Example output:

  23 45 67
  44 55 0
  0 0 0
  255 255 255

Data Constructs - Structures

- Exercise: write a program to
  - read a file of integer coordinates and RGB values
  - store the input in an array of points
  - delete any duplicate points (keep the first point)
  - write out the coordinates and RGB values to an output file
  - the input file is points.txt
  - the output file is new_points.txt

Data Constructs - Structures

- Example input:

  1 2 23 45 67
  3 3 44 55 0
  3 3 0 0 0
  4 4 255 255 255

- Example output:

  1 2 23 45 67
  3 3 44 55 0
  4 4 255 255 255

Arrays, Pointers, and Strings
Address Arithmetic

- "Address of" operator `&`

  The value of an expression `&x` is an address

- Other expressions yield addresses
  - the name of an array, written without brackets
  - the address is that of the first element of the array

    ```
    char s[50];
    ```

    `s` is equivalent to `&(s[0])`

  - we can combine the name of an array with integers

    `s` is equivalent to `&(s[0])`

    `s+i` is equivalent to `&(s[i])`

Arrays, Pointers, and Strings
Address Arithmetic

- Such expressions are valid even when the array elements are not 1 byte in size

- In general address arithmetic takes into account the size of the element

```
int a[10];
```

`a+i` is equivalent to `&(a[i])`

  - Such a capability leads some people to write:

```
for (i=0; i<10; i++) scanf("%d", a+i);
```

rather than

```
for (i=0; i<10; i++) scanf("%d", &a[i]);
```

Arrays, Pointers, and Strings
Address Arithmetic

- Indirection operator `*`

  The value of an expression such as `*a` is the object to which the address `a` refers

  `*a` is equivalent to `a[0]`

  `*(a+i)` is equivalent to `a[i]`

Arrays, Pointers, and Strings
Function Arguments and Arrays

- In C and C++ there is no need for special parameter-passing for arrays
- We pass the address of the first element of the array
- Which is the array name!
- We automatically have access to all other elements in the array

Functions
Function Arguments and Arrays

```
// MINIMUM: finding the smallest element of an
// integer array

#include <iostream.h>

int main() {
   int minimum(int *a, int n);
   int table[10];

   cout << "Enter 10 integers: \n";

   for (int i=0; i<10; i++) cin >> table[i];
   cout << "\nThe minimum of these values is "
        << minimum(table, 10) << endl;
   return 0;
}
```

Functions
Function Arguments and Arrays

```
// definition of minimum, version A

int minimum(int *a, int n)
{  int small = *a;
   for (int i=1; i<n; i++)
      if (*(a+i) < small)
         small = *(a+i);
   return small;
}
```

Functions
Function Arguments and Arrays

```
// definition of minimum, version B (for Better!)

int minimum(int a[], int n)
{  int small = a[0];
   for (int i=1; i<n; i++)
      if (a[i] < small)
         small = a[i];
   return small;
}
```

Arrays, Pointers, and Strings
Pointers

- In the following `p` is a pointer variable

```
int *p, n=5, k;
```

- Pointers store addresses

```
p= &n;
k = *p  // k is now equal to???
```

  - `*` is sometimes known as a **dereferencing operator** and accessing the object to which the pointer points is known as dereferencing

```
p = &n
```

Arrays, Pointers, and Strings
Pointers

It is essential to assign value to pointers

after declaring `p` we must not use `*p` before assigning a value to `p`.

```
int main()
{   char *p, ch;
    *p = 'A'; // Serious error! Why?
    return 0;
}
```

Arrays, Pointers, and Strings
Pointers

It is essential to assign value to pointers

after declaring `p` we must not use `*p` before assigning a value to `p`.

```
int main()
{   char *p, ch;
    p = &ch;
    *p = 'A';
    return 0;
}
```

Arrays, Pointers, and Strings
Pointers

Pointer conversion and void-pointers

```
int i;
char *p_char;

p_char = &i; // error: incompatible types
             // pointer_to_char and
             // pointer_to_int

p_char = (char *)&i;
             // OK: casting pointer_to_int
             // as pointer_to_char
```

Arrays, Pointers, and Strings

Pointers

In C++ we have generic pointer types: void_pointers

```
int i;
char *p_char;
void *p_void;

p_void = &i; // pointer_to_int to pointer_to_void
p_char = (char *)p_void;
             // cast needed in C++ (but not ANSI C)
             // for pointer_to_void to
             // pointer_to_int
```

Arrays, Pointers, and Strings
Pointers

- void_pointers can be used in comparisons

```
int  *p_int;
char *p_char;
void *p_void;

if (p_char == p_int) ... // Error
if (p_void == p_int) ... // OK
```

- Address arithmetic must not be applied to void_pointers. Why?

Arrays, Pointers, and Strings
Pointers

Typedef declarations

used to introduce a new identifier denote an (arbitrarily complex) type

```
typedef double real;
typedef int *ptr;
...
real x,y;    // double
ptr p;       // pointer_to_int
```

Arrays, Pointers, and Strings

Pointers

Initialization of pointers

```
int i, a[10];
int *p = &i;  // initial value of p is &i
int *q = a;   // initial value of q is the
              // address of the first element
              // of array a
```

Arrays, Pointers, and Strings
Strings

- Recap: addresses can appear in the following three forms
  - expression beginning with the & operator
  - the name of an array
  - Pointer

- Another, fourth, important form which yields an address
  - A string (string constant or string literal)
  - “ABC”

Arrays, Pointers, and Strings

Strings

“ABC”

- effectively an array with four char elements:
- ‘A’, ‘B’, ‘C’, and ‘\0’
- The value of this string is the address of its first character and its type is pointer_to_char

```
*”ABC”          is equal to   ‘A’
*(“ABC” + 1)    is equal to   ‘B’
*(“ABC” + 2)    is equal to   ‘C’
*(“ABC” + 3)    is equal to   ‘\0’
```

Arrays, Pointers, and Strings

Strings

"ABC"

- effectively an array with four char elements:
- 'A', 'B', 'C', and '\0'
- The value of this string is the address of its first character and its type is pointer_to_char

```
"ABC"[0]          is equal to    'A'
"ABC"[1]          is equal to    'B'
"ABC"[2]          is equal to    'C'
"ABC"[3]          is equal to    '\0'
```

Arrays, Pointers, and Strings
Strings

Assigning the address of a string literal to a pointer variable can be useful:

```
// POINTER
#include <stdio.h>
int main()
{   char *name = “main”;
    printf(name);
    return 0;
}
```

```
// POINTER
#include <iostream.h>
int main()
{   char *name = “main”;
    cout << name;
    return 0;
}
```

Arrays, Pointers, and Strings
Strings Operations

Many string handling operations are declared in string.h

```
#include <string.h>

char s[4];

s = “ABC”; // Error: can’t do this in C; Why?
strcpy(s, “ABC”);  // string copy
```

Arrays, Pointers, and Strings
Strings Operations

Many string handling operations are declared in string.h

```
#include <string.h>
#include <iostream.h>

int main()
{   char s[100]="Program something.", t[100];
    strcpy(t, s);
    strcpy(t+8, "in C++.";
    cout << s << endl << t << endl;
    return 0;
}   // what is the output?
```

Arrays, Pointers, and Strings
Strings Operations

Many string handling operations are declared in string.h

```
strlen(string);
   // returns the length of the string

E.g.
int length;
char s[100]="ABC";
length = strlen(s); // returns 3
```

Arrays, Pointers, and Strings
Strings Operations

Many string handling operations are declared in string.h

```
strcat(destination, source);
   // concatenate source to destination

strncat(destination, source, n);
   // concatenate n characters of source
   // to destination
   // programmer is responsible for making
   // sure there is enough room
```

Arrays, Pointers, and Strings
Strings Operations

Many string handling operations are declared in string.h

```
strcmp(string1, string2);
   // returns 0 in the case of equality
   // returns <0 if string1 < string2
   // returns >0 if string1 > string2

strncmp(string1, string2, n);
   // same as strcmp except only n characters
   // considered in the test
```

Arrays, Pointers, and Strings
Dynamic Memory Allocation

- Array declarations
  - require a constant length specification
  - cannot declare variable length arrays

- However, in C++ we can create an array whose length is defined at run-time

```
int n;
char *s;
...
cin >> n;
s = new char[n];
```

Arrays, Pointers, and Strings
Dynamic Memory Allocation

```
// TESTMEM: test how much memory is available
#include <iostream.h>
#include <new.h> // may not be required ... must check

int main() {
   char *p;
   for (int i=1;;i++){      // horrible style ☹
      p = new char[10000]; // intentional memory leakage
      if (p == 0) break;
      cout << "Allocated: " << 10 * i << "kB\n";
   }
   return 0;
}                           // rewrite in a better style!
```

Arrays, Pointers, and Strings
Dynamic Memory Allocation

Memory is deallocated with `delete()`

- `p = new int // deallocate with:`
- `delete p;`

- `p = new int[m] // deallocate with:`
- `delete[] p;`

- `delete` is only available in C++

Arrays, Pointers, and Strings
Dynamic Memory Allocation

`malloc()`

- standard C memory allocation function
- declared in stdlib.h
- its argument defines the number of bytes to be allocated

```
#include <stdlib.h>
int n;
char *s;
...
cin > n;
s = (char *) malloc (n);
```

Arrays, Pointers, and Strings
Dynamic Memory Allocation

`malloc()`

- but to allocate an array of floats:

```
#include <stdlib.h>
int n;
float *f;
...
cin > n;
s = (float *) malloc (n * sizeof(float));
```

- `malloc()` returns `NULL` if allocation fails

Arrays, Pointers, and Strings
Dynamic Memory Allocation

malloc()

```
s = (float *) malloc (n * sizeof(float));
if (s == NULL)
{  cout << "Not enough memory.\n";
   exit(1); // terminates execution of program
}           // argument 1: abnormal termination
```

Arrays, Pointers, and Strings
Dynamic Memory Allocation

`calloc()`

- Takes two arguments
  - number of elements
  - size of each element in bytes
- all values are initialized to zero
- `calloc()` returns `NULL` if allocation fails

Arrays, Pointers, and Strings
Dynamic Memory Allocation

Memory is deallocated with `free()`

```
free(s);
```

Arrays, Pointers, and Strings
Input and Output of Strings

Input

```
char[40] s;
...
scanf("%s", s);  // skips whitespace and terminates on
                 // whitespace
cin >> s;        // same as scanf
gets(s);         // reads an entire line

// problems if more than 40 chars are typed:
// ABCDEFGHIJKLMNOPQRSTUVWZYZabcedfghijklmnopqrstuvwzyz
// requires a string of 53 elements
```

Arrays, Pointers, and Strings
Input and Output of Strings

Input

```
char[40] s;
...
scanf(“%39s”, s);       //reads at most 39 characters
cin >> setw(40) >> s; // same as scanf
fgets(s, 40, stdin);  // reads a line of at most 39
                      // characters, including \n
cin.getline(s, 40);   // reads a line of at most 39
                      // characters, including \n
                      // but doesn’t put \n in s
```

Arrays, Pointers, and Strings
Input and Output of Strings

Output

```
char[40] s;
...
printf(s);              // Display just the contents of s
printf("%s", s);        // same
cout << s;              // same
printf("%s\n", s);      // Display s, followed by newline
puts(s);                // same
```

Arrays, Pointers, and Strings
Input and Output of Strings

Output

```
// ALIGN1: strings in a table, based on standard I/O
#include <stdio.h>

int main()
{  char *p[3] = {“Charles”, “Tim”, “Peter”};
   int age[3] = {21, 5, 12}, i;
   for (i=0; i<3; i++)
      printf(“%-12s%3d\n”, p[i], age[i]); // left align
   return 0;
}
```

Arrays, Pointers, and Strings
Input and Output of Strings

Output

```
// ALIGN2: strings in a table, based on stream I/O
#include <iostream.h>
#include <iomanip.h>
int main()
{  char *p[3] = {“Charles”, “Tim”, “Peter”};
   int age[3] = {21, 5, 12}, i;
   for (i=0; i<3; i++)
     cout << setw(12) << setiosflags(ios::left) << p[i]
          << setw(3) < resetiosflags(ios::left)
          << age[i];
return 0;
}
```

Arrays, Pointers, and Strings

Multi-Dimensional Arrays

A table or matrix

- can be regarded as an array whose elements are also arrays

```
float table[20][5]
int a[2][3] = {{60,30,50}, {20,80,40}};
int b[2][3] = {60,30,50,20,80,40};
char namelist[3][30]
= {"Johnson", "Peterson", "Jacobson");
...
for (i=0; i<3; i++)
    cout << namelist[i] << endl;
```

Arrays, Pointers, and Strings

Multi-Dimensional Arrays

Pointers to 2-D arrays:

```
int i, j;
int a[2][3] = {{60,30,50}, {20,80,40}};
int (*p)[3]; // p is a pointer to a 1-D array
             // of three int elements
...
p = a; // p points to first row of a
a[i][j] = 0;        // all four statements
(*(a+i))[j] = 0;    // are equivalent
p[i][j] = 0;        // remember [] has higher
(*(p+i))[j] = 0;    // priority than *
```

Arrays, Pointers, and Strings

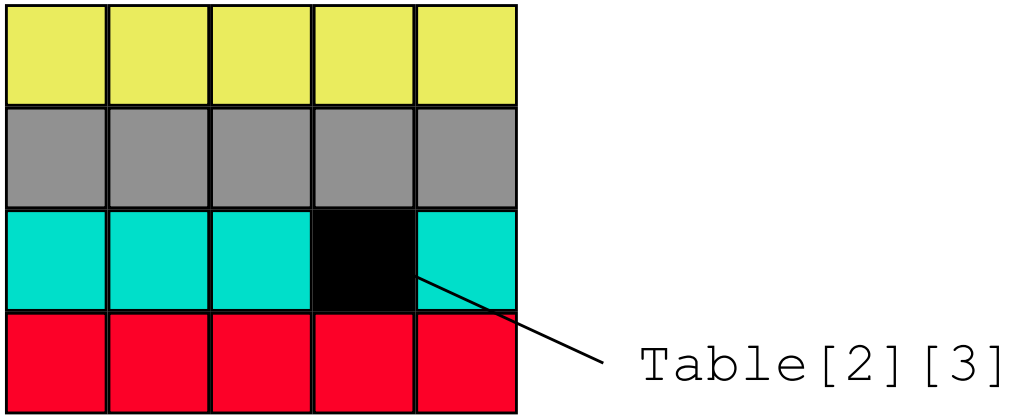
Multi-Dimensional Arrays

Function Parameters

```
int main()
{   float table[4][5];
    int f(float t[][5]);

  f(table);
    return 0;
}
int f(float t[][5]) // may omit the first dimension
{                   // but all other dimensions must
}                   // be declared since it must be
                    // possible to compute the
                   // address of each element. How?
```

Arrays, Pointers, and Strings

Multi-Dimensional Arrays



The address of Table[i][j] is computed by the mapping function 5*i + j (e.g. 5*2+3 = 13)

Arrays, Pointers, and Strings

Multi-Dimensional Arrays

Arrays of Pointers

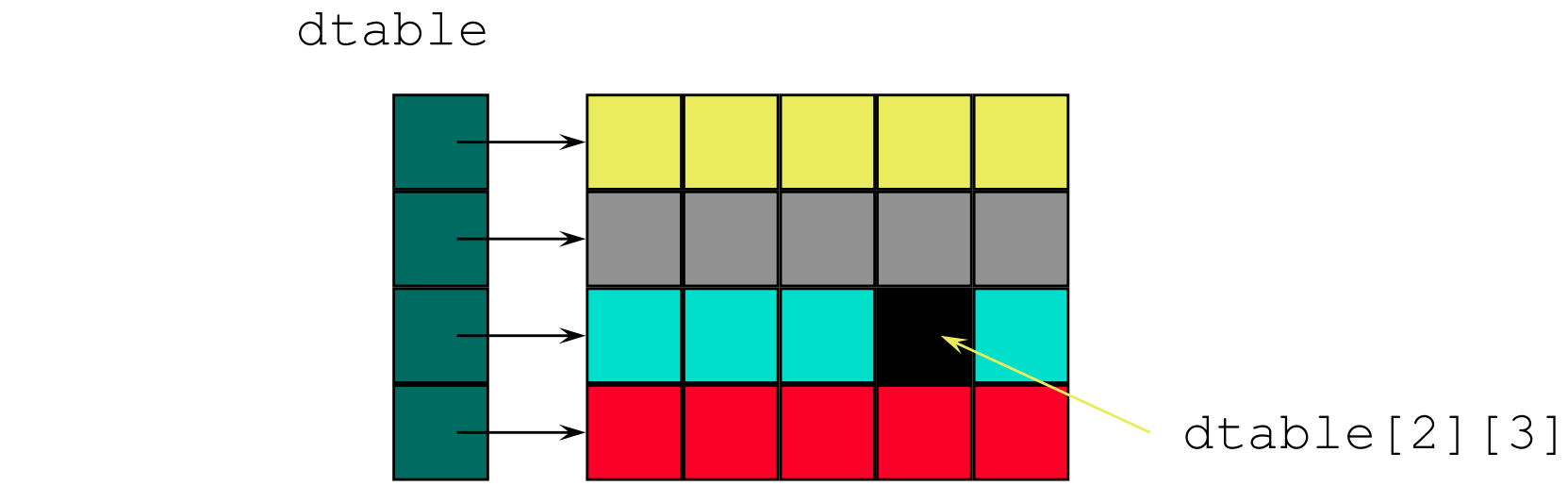
we can create 2-D 'arrays' in a slightly different (& more efficient) way using

- an array of pointers to 1-D arrays, and
- a sequence of 1-D arrays

```
float *dtable[4]; // array of 4 pointers to floats

for (i=0; i<20; i++)
{  dtable[i] = new float[5];
   if (dtable[i] == NULL)
   {  cout << " Not enough memory"; exit(1);
   }
}
```

Arrays, Pointers, and Strings

Multi-Dimensional Arrays



dtable[i][j] is equivalent to (*(dtable+i))[j] ...
there is no multiplication in the computation
of the address, just indirection.

Arrays, Pointers, and Strings

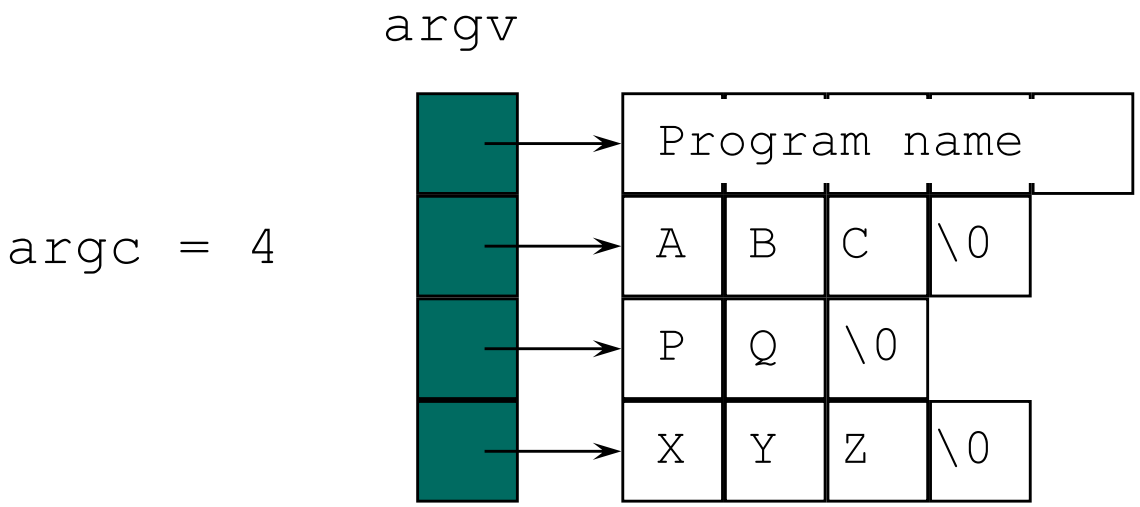
Program Parameters

The main() function of a program can have parameters

- called program parameters
- an arbitrary number of arguments can be supplied
- represented as a sequence of character strings
- two parameters
  - argc ... the number of parameters (argument count)
  - argv ... an array of pointers to strings (argument vector)

Arrays, Pointers, and Strings

Program Parameters



Program parameters for an invocation of the form

program ABC PQ XYZ

Arrays, Pointers, and Strings

Program Parameters

```
// PROGPARAM:  Demonstration of using program parmeters
#include <iostream.h>

int main(int argc, char *argv[])
{   cout << "argc  = " << argc << endl;
    for (int i=1; i<argc; i++)
        cout << "argv[" << i << "]= " << argv[i] << endl;
    return 0;
}
```

Arrays, Pointers, and Strings

In-Memory Format Conversion

`sscanf()`

- scans a string and converts to the designated type

```
#include <stdio.h>
...
char s[50]="123   456 \n98.756";
int i, j;
double x;
sscanf(s, "%d %d %lf", &i, &j, &x);
```

- sscanf returns the number of value successfully scanned

Arrays, Pointers, and Strings

In-Memory Format Conversion

`sprintf()`

- fills a string with the characters representing the passed arguments

```
#include <stdio.h>
...
char s[50]="123   456 \n98.756";
sprintf(s,"Sum: %6.3f Difference:%6.3f",
            45 + 2.89, 45 - 2.89);
```

Arrays, Pointers, and Strings

Pointers to Functions

In C and C++ we can assign the start address of functions to pointers

```
// function definition
float example (int i, int j)
{  return 3.14159 * i + j;
}

float (*p)(int i, int j); // declaration
...
p = example;
```

Arrays, Pointers, and Strings

Pointers to Functions

- And we can now invoke the function as follows

```
(*p)(12, 34); // same as example(12,34);
```

- We can omit the * and the () to get:

```
p(12, 34);    // !!
```

- Pointers to function also allow us to pass functions as arguments to other functions

Arrays, Pointers, and Strings

Exercise

11. Write and test a function to

- read a string representing a WWW URL (e.g. http://www.vernon.eu)
- replace the // with \\
- write the string back out again

Arrays, Pointers, and Strings

Exercises

12. Write a interactive user interface which allows a user to exercise all of the set operators for three pre-defined sets A, B, and C

    Commands should be simple single keywords with zero, one, or two operands, as appropriate

- add A 10
- union C A B
- list A
- intersection C A B
- remove 1 B