C

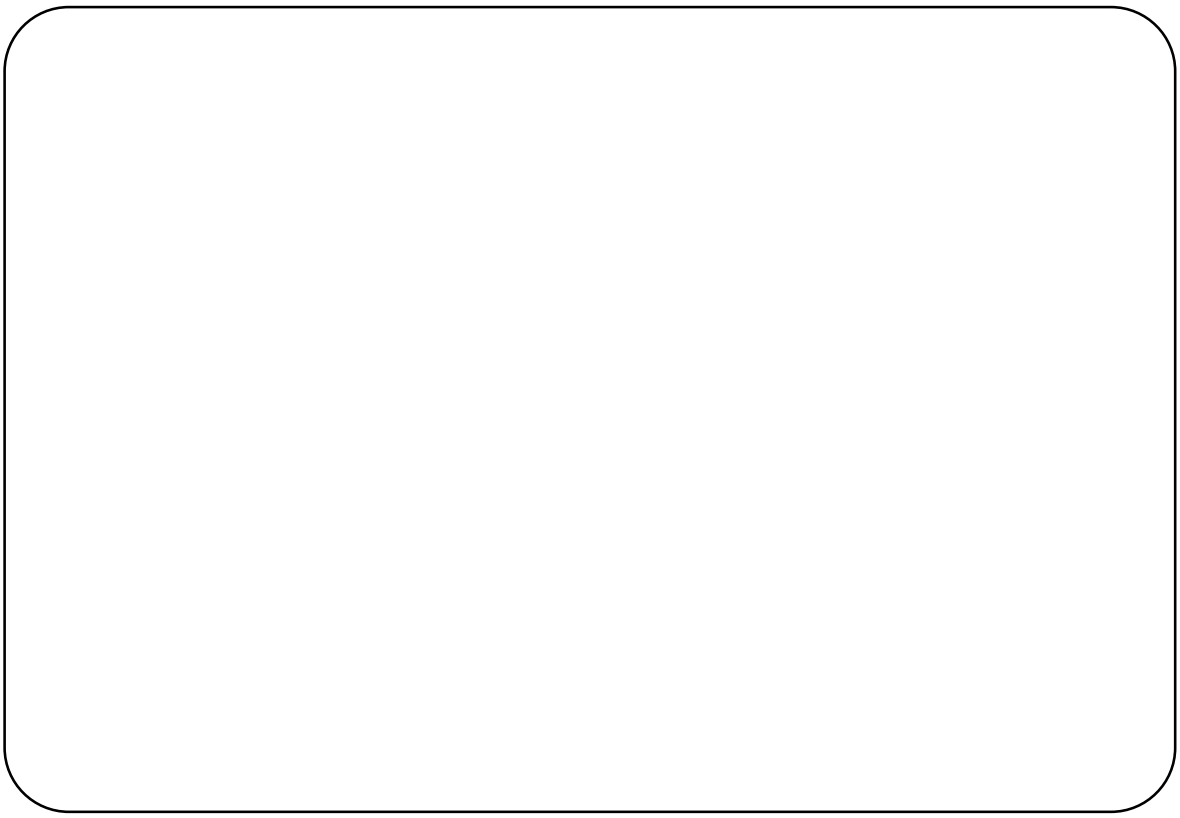
Programming

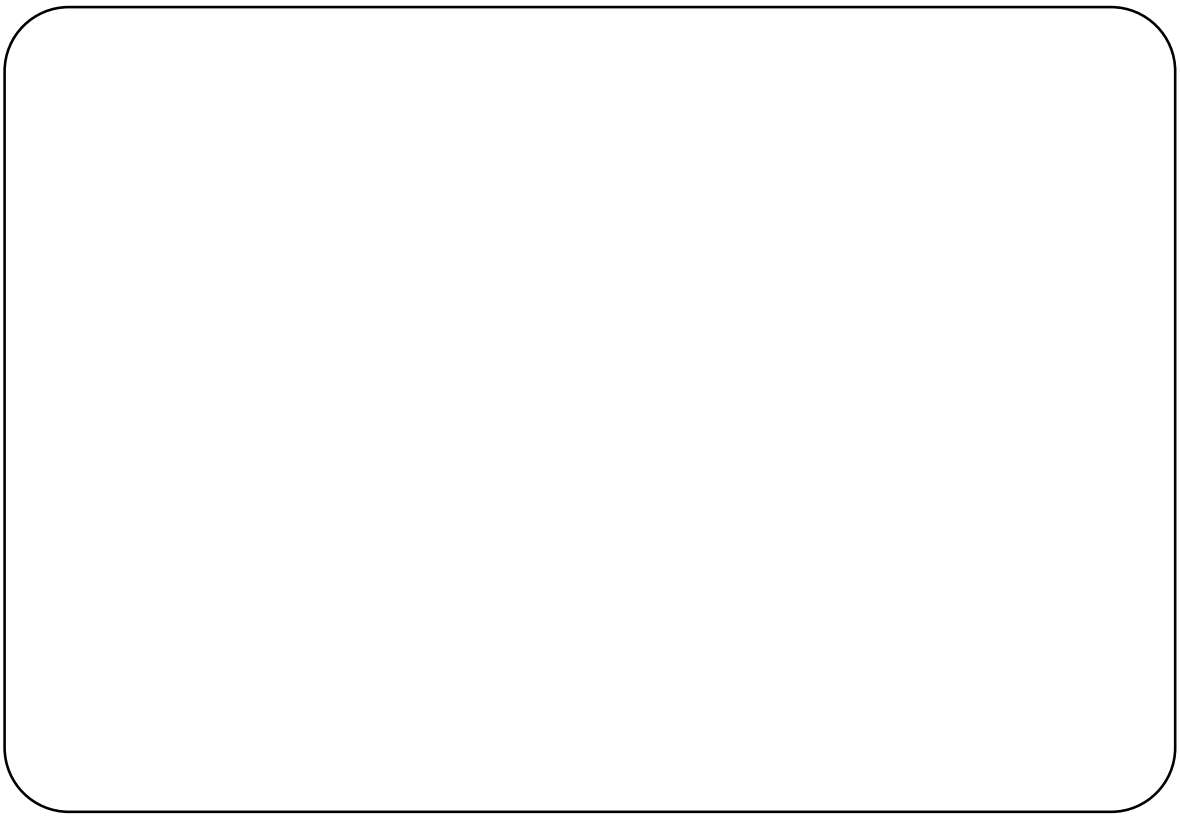
by

Chris Seddon

seddon-software@keme.co.uk

C Programming





Introduction

- **The Language**
  - History and Evolution of C
  - Characteristics of C
- **C Programs**
  - Program structure
  - First Program
  - Simple I/O C

1

© CRS Enterprises Ltd 2001-15

History and Evolution of C

- **C is a general-purpose, procedural programming language**
  - **developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories**
  - **original for use with the Unix operating system**
- **Standardised in 1989**
  - **known as C89**
  - **revised in 1999**
- **Has influenced many other languages**
  - **especially C++**
  - **also Java, C#**

C is a general-purpose, procedural programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories. It was originally used to write the Unix kernel, but quickly became the most popular programming language. In the 70s and early 80s many variants of C appeared; every machine had its own version (only the Unix versions seemed to be controlled in any way). In the late 80s the American National Standards Institute (ANSI) Committee revamped and standardised the language.

The C standard is now accepted in all countries that subscribe to ISO, the International Standards Organisation. The C standard is now known as BS EN ISO 9899 (C89 for short). In 1994, work began on the update to ISO 9899; culminating in a revised standard published in 1999. This new standard is poorly supported and most applications are continuing to use the C89 standard.

What is C?

- Clean, small, expressive language
- Structured procedural programming
- Separate compilation of modules
- Highly-efficient object code
- Ease of access to hardware features
- Portable
- Extensive library of functions

- A 'write-only' language reputation!!!

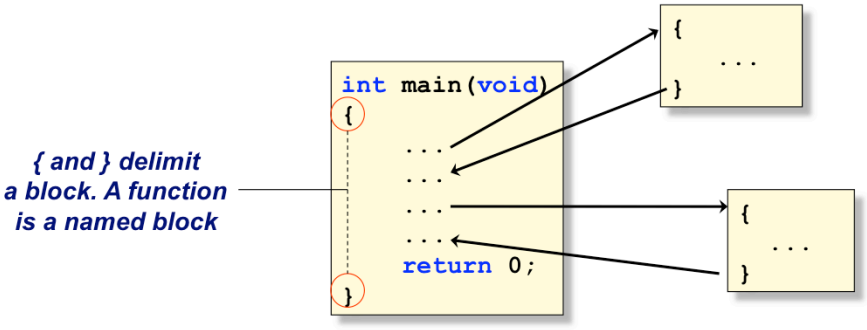
The characteristics of the language listed above are fairly self-explicit. The language itself is small and easy to learn, but it lacks the object oriented support common to modern programming languages.

Constructs within the language lend themselves quite naturally to small and fast executable code and nowadays C is often used in conjunction with other languages such as Python, Java and C# to code the subsystems that need to run fast.

The language's 'write-only' reputation was gained during C's infancy, when systems programmers and blatant hackers(!) used C's free-style formatting and terse syntax to the full. This reputation is fast losing ground as C coding standards have emerged, and better design practices have been adopted.

Program Structure

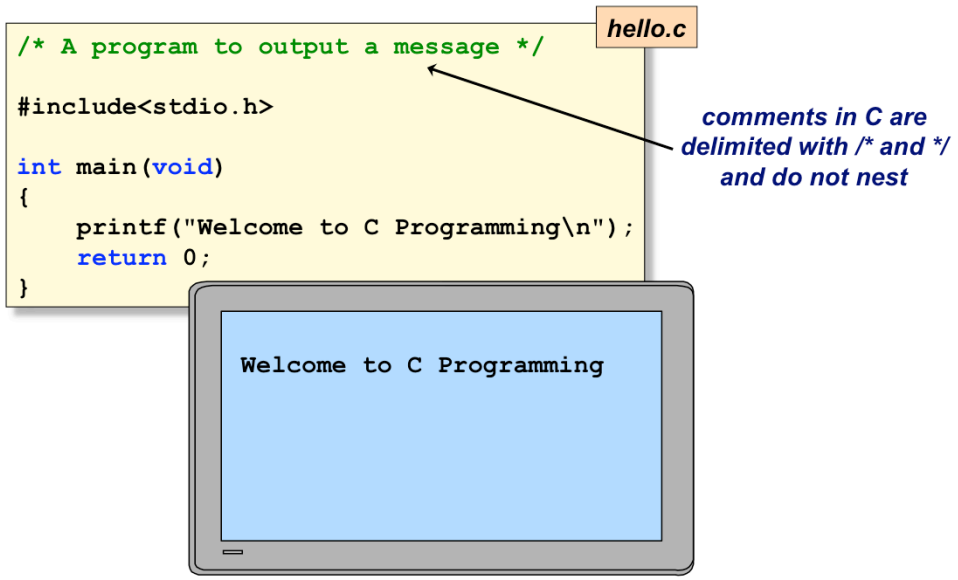
- **A program consists of one or more functions**
  - **execution begins at the function called 'main'**
  - **execution continues by calling...**
    - **user-written functions**
    - **library functions**



A C program consists of one or more functions which can be written across multiple source files. Execution begins at the function called main and continues by calling user-written and library functions.

Each file is called a compilation module, because files are compiled one at a time. The linker is used to combine the output of all the compilation modules and library modules to form the final executable.

Traditional First Program



 9

The example is the classic first program. It just has a main function that prints a message to STDOUT.

The include file is required to define the printf library function and the '\n' character sequence represents the newline character.

The example also includes a C multi-line comment, delimited by the character-pairs / * and */. Many C compilers nowadays allow use of the C++ style // comments even though it is not part of the standard.

Adding Two Numbers

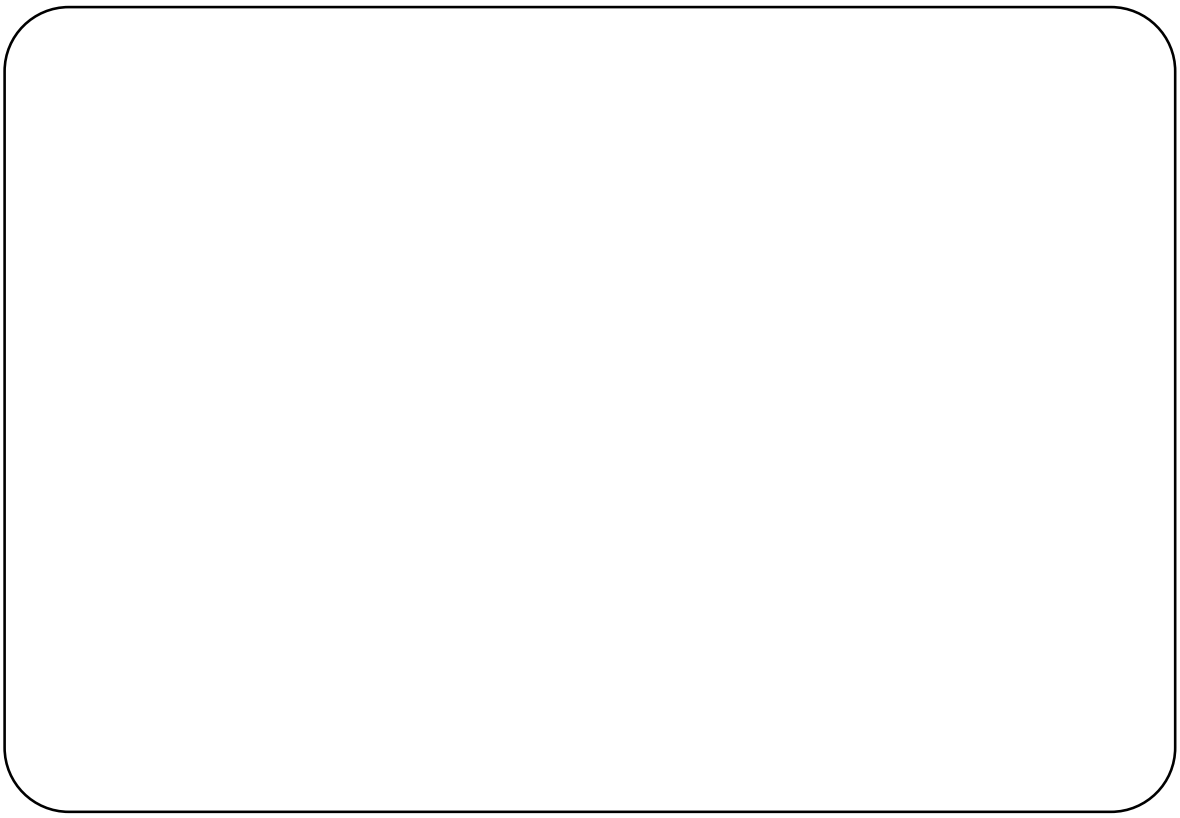
numbers.c

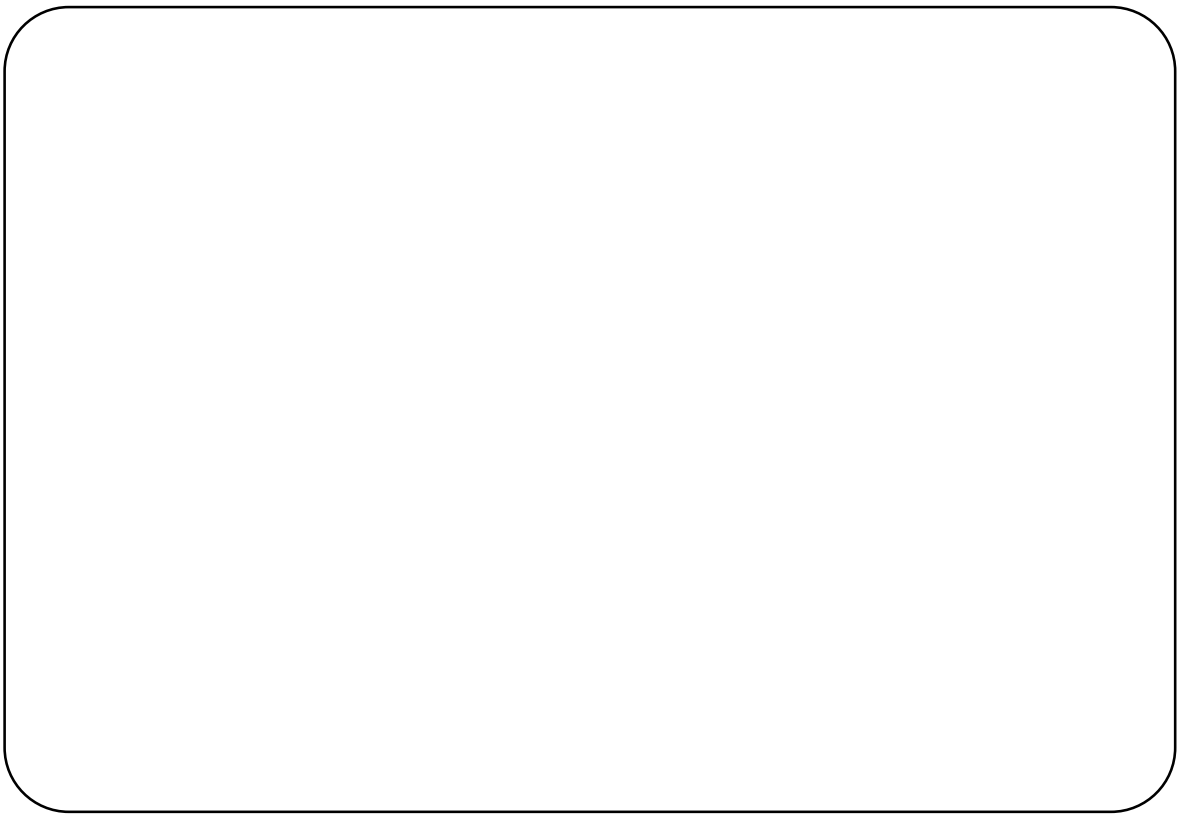
```
#include<stdio.h>

int main(void)
{
    int x, y, result;
    printf("Please input two numbers: \n");
    scanf("%i %i", &x, &y);
    result = x + y;
    printf("The sum of %i and %i is %d\n", x, y, result);
    return 0;
}
```

printf - formatted output
scanf - formatted input

The program example illustrates more extensive program. The main function now includes data as well as code statements. The first line is a data statement, which declares three data items: x, y and result defined as integers. The remaining statements manipulate this data.

The printf statement writes to STDOUT; the scanf function reads from STDIN. Both these functions use a format string as their first argument (hence the 'f' in printf and scanf). The % characters are place holders for the data being written or read. Thus, the two %i in scanf are paired off with &x and &y respectively. An explanation of the & operator is covered in the pointers chapter.





Data Types

- **Objective**
  - Explore C's built-in scalar types
- **Contents**
  - Variable declaration format
  - Rules for legal identifiers
  - Scalar types
  - Initialising variables
  - Constants

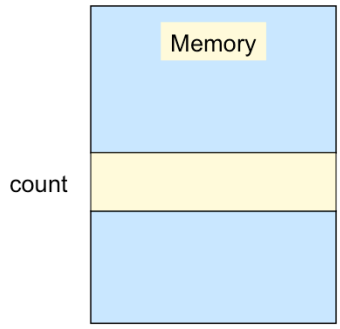
2

© CRS Enterprises Ltd 2001-15

Variable Declarations

- **A variable declaration reserves an area of memory and gives it a symbolic name**
- **The type determines the amount of memory reserved and the values that it can contain**

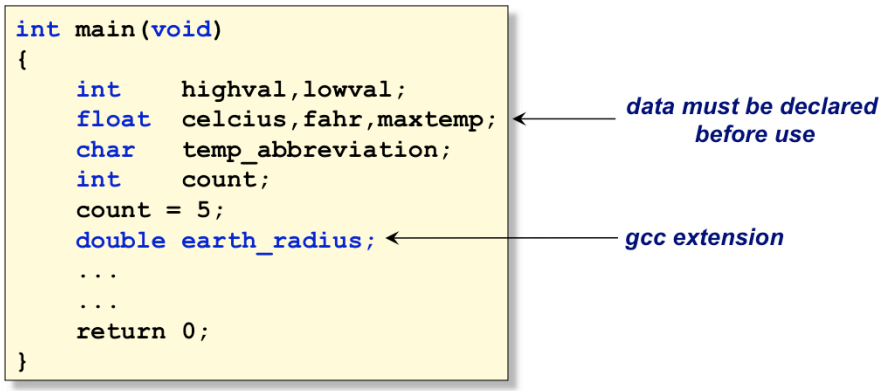
```
int main(void)
{
    int count;
    ...
    return 0;
}
```



All data must be declared before it can be used. This is achieved using a variable declaration; the declaration reserves an area of memory and gives it a symbolic name. Every variable must be associated with a type; the type determines the amount of memory reserved and the values that it can contain. Variables cannot change type during the execution of a program - we say that C is a strongly typed language.

General Declaration Format

- **A declaration consists of**
  - **a type**
  - **a comma separated list of variables of that type**



You can declare several variables for a given type all at once - use a comma separated list. All variables must be declared at the top of a block, before first use. This is not true in C++ where variables can be declared anywhere in a block (but before first use) and several C compilers (such as gcc) allow you to follow the C++ convention. There is no doubt that this can be very convenient, but it is not part of the C89 standard.

Identifiers

- **C identifiers**
  - declare before use
  - CaSe sensitive
  - letters, digits, underscore
  - start with a letter or underscore

```
typeless = 0;
```

```
int typed = 0;
```

```
int 43answer;
```

```
int different = 0;
int Different = 0;
```

```
int answer_42;
```

underscore is considered a letter

- **C keywords**
  - reserved
  - cannot be used as identifiers

```
int double ...
return if else ...
```

 16

C uses identifiers for variables and function names. Each identifier is case sensitive and consists of letters, digits, underscore. A identifier must start with a letter or underscore, not with a digit. The C compiler often uses a double underscore for its own identifiers, so try to avoid identifiers starting with two underscores. Many programmers use a leading or trailing underscore in identifiers.

C keywords are reserved and cannot be used as identifiers. If you are not sure which identifiers are reserved, don't worry, the compiler will soon let you know!

Fundamental Types

- **character types**
  - char (8 bit Ascii)
  - wchar_t (16 bit Unicode)
- **integral types**
  - short, int, long
  - signed, unsigned
  - usually 32/64 bit
- **decimal types**
  - float, double, long double
  - float has 7 sig. digits
  - double has 13 sig. digits

```
char    c1 = 'a';
wchar_t c2 = L'a';

int   x1 = 5000;
short x2 = 2
long  x3 = 6712354L;

float  n1 = 45.7134F;
double n2 = 681.7312;
```


C has a limited number of built in types: they are classified as character, integral and decimal types. In practice this does not turn out to be a problem - as you will find out later, you can define your own types in C.

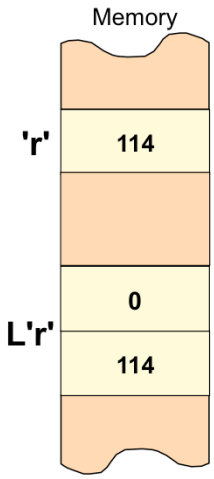
Character types were originally restricted to using the Latin character set and hence were defined as 8 bit characters. Nowadays, internationalisation means we must cater for many character sets such as those used in Arabic, Indian and Chinese scripts. C allows the wide character type (Unicode) to define characters in these scripts.

Several integral types are defined; the most important of which is int. This type nearly always has the same size as the underlying word size of the OS. Thus an int is 32 on most operating systems, but with the advent of 64 bit Windows and Unix this is set to change. To save space you can use short or to gain precision you can use long. Both signed and unsigned types are supported. In C89, the sizes of integral types are compiler dependent, but in C98 you can define fixed size integral types.

Decimal types also come in a variety of types. In modern applications you should avoid float because of its lack of precision. double suffices for most applications.

Character Types

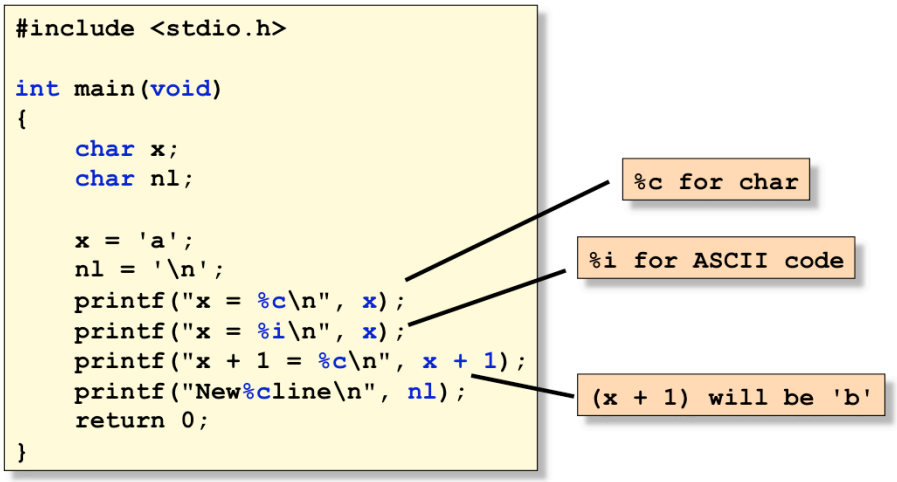
- **char is used to store a single character**
  - **its value is invariably the ASCII code**
  - **use single quotes**
  - **usually 1 byte**
- **wchar_t is for Unicode**
  - **16 bit characters**
  - **Arabic, Greek, Japanese, Chinese ...**
  - **Unicode Standard**



char is used to store a single character and although a compiler can support any internal character set, nowadays its value is invariably an ASCII code. Characters are defined using single quotes; double quotes are reserved for strings (character arrays). Virtually every compiler uses 1 byte per character.

wchar_t is used for Unicode (16 bit) characters. Arabic, Greek, Japanese, Chinese and many other character sets are supported using the Unicode Standard (ISBN 0-321-18578-1). To distinguish a Unicode character from a normal character use the prefix L. Note that the first 256 characters in Unicode are the Latin characters (hence the leading zero in the definition of Unicode L'r').

Example



This example uses printf to output a character variable - use %c in the format string. Note that characters can in some sense be considered as integers. If you print a character using %i you will reveal its underlying ASCII representation. Not only that, but you can add one to a character to get the next character in the set. In ASCII this means an 'A' will become a 'B'; a 'B' will become a 'C' and so on.

Integral Types

- **Types**
  - short, int, long, signed and unsigned
- **Storage**
  - short int <= int
  - int <= long int
  - short must be at least 16 bits long
  - long must be at least 32 bits long
- **C99 has additional types**
  - fixed size ints
  - booleans
  - complex number

```
short     s1 = 2;
short int s2 = 2;

int          i = 5000;
unsigned int i = 5000;
signed int   i = 5000;

long  x3 = 6712354L;
long  x3 = 6712354L;
```

 20

Integral types follow the following rules in C89:

short int <= int

int <= long int

short must be at least 16 bits long

long must be at least 32 bits long

Note that the actual storage size of these types is not defined by these rules and are compiler dependent.

C99 has additional types to facilitate portability: you can define 16, 32 and 64 bit integral types. C89 does not have a boolean type and integers are used instead (see later). C98 addresses this deficiency and also introduces a complex type. Unfortunately, the complex type is poorly supported in practice and is badly broken in gcc.

Floating Point Numbers

- **Floating point types hold fractional and large numbers**
  - **float holds a single precision floating point number**
  - **double holds a double precision floating point number**
  - **long double provides even greater accuracy and precision**
- **Modern code uses double**
  - **use postfix F or f to define a literal constant if single precision is necessary**

```
int main(void)
{
    double jeopardy = 3.75;
    float  away = 3.75F;
    ...
    return 0;
}
```

```
3.14159265359;
-273.15
4.0
0.0
273.
.15
17.5F
299792.458
3e8f
6.672E-11F
```

3×10^8

6.672×10^{-11}

Use floating point types hold fractional and large numbers. Avoid using float because it has limited precision and can lead to significant rounding errors. Most programs use double for all floating point numbers, but long double is available to provide even greater accuracy and precision if required.

Exercise

- **Which names are legal in the code fragment below?**

```
int main(void)
{
    char  code;
    int   employee_number, 999number, age-in-1986;
    int   EMPLOYEE_DEPT, emp_count;
    float _salary, overtimeHours;
    char  cEmpCode;
    int   double;
    ...
    return 0;
}
```

 22

Most of the identifiers above are legal. The identifiers that you can't use are

999number	- begins with a digit
age-in-1986	- hyphen is not permitted
double	- double is a keyword

Note that EMPLOYEE_DEPT is allowed, but discouraged; identifiers are usually in lower case with subsequent words capitalised (overtimeHours) or separated by underscores (emp_count). Note that Microsoft (but none else!) recommend Hungarian notation, where the leading characters of the identifier indicate its type (cEmpCode). This was always a poor convention and is now falling into disuse.

Initialising Variables

- **Variables may be initialised to specific values in their declarations**
- **Global variables default to zero**
- **Local variables not initialised automatically**

```
int main(void)
{
    int    i = 0;
    int    mins_in_day = 24 * 60;
    double base_rate = 3.75;
    char   letter_b = 'b';
    ...
    return 0;
}
```


Variables may be initialised to specific values in their declarations. Because of the way storage is allocated by C compilers, you will find that all initialised global variables will default to zero, but local variables will not be initialised automatically - they will have an arbitrary value (whatever happens to be in memory when their storage is allocated).

Constants

- **Constants must be initialised at declaration time**
  - **r-value**
  - **compiler forbids assignment**

```
int main(void)
{
    const int    MONTHS = 12;
    const double PI = 3.14159;
    const char   SPACE = ' ';
    ...
    return 0;
}
```

- **Older style uses the preprocessor**

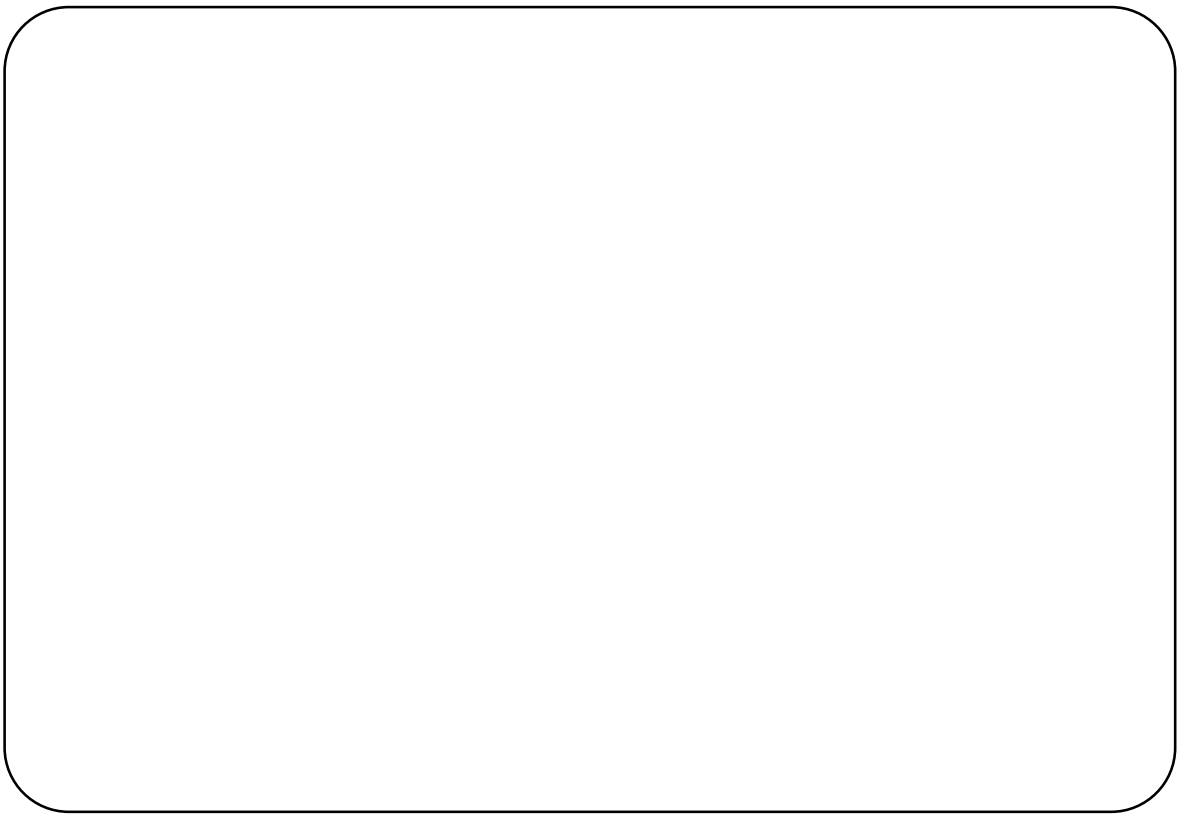
no semicolon

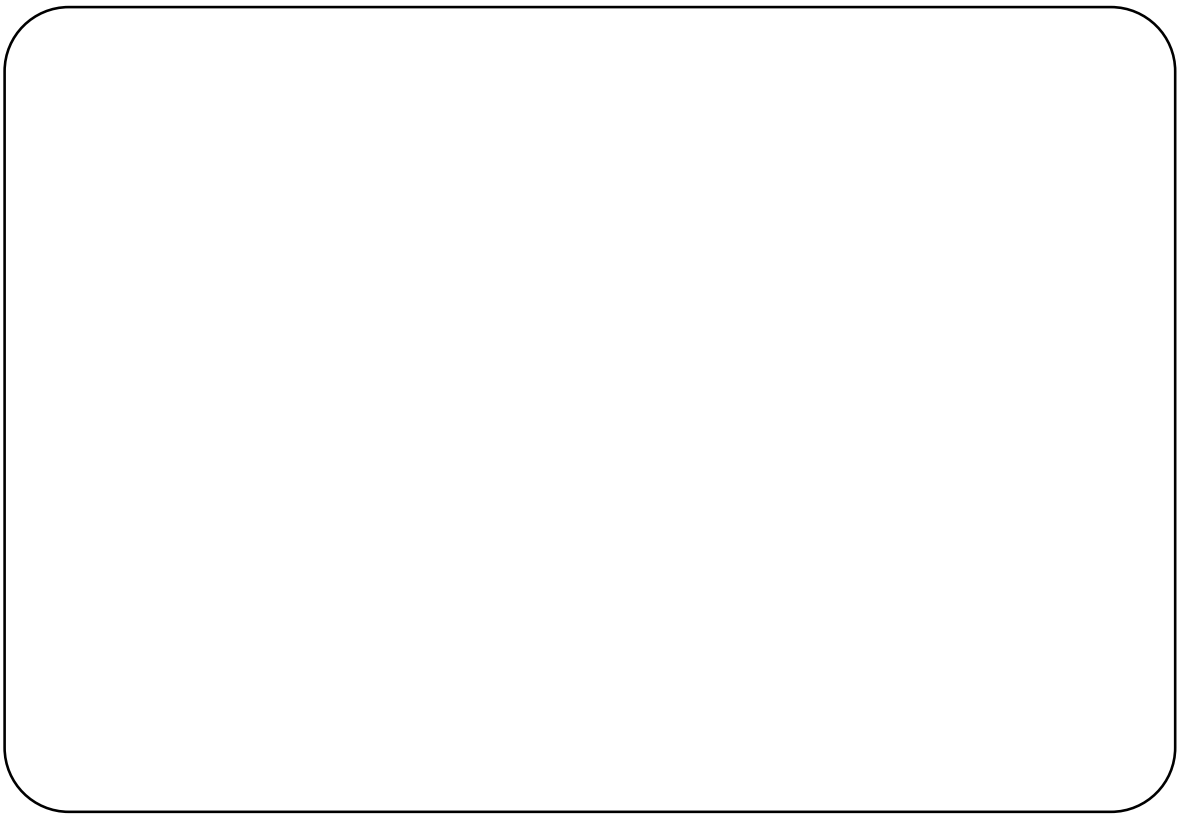
```
#define PI = 3.14159
```


Constants can be defined in C programs using the const qualifier. The compiler will ensure constants are not modified after definition, by only permitting the use of a constant in the right hand side of an assignment (hence the term r-value). Variables can be placed on the left hand side of an assignment (hence the term l-value).

A common convention is to use block capitals for constants - to make them stand out.

Early C compilers did not use the const keyword and constants had to be defined in the pre-processor. Note that if you use the pre-processor to define a constant do not terminate the definition with a semi-colon (the pre-processor is not C!). The pre-processor will be discussed later in the course.





Expressions

- Basic Operators
  - Binary operators
  - Logical operators
  - Bitwise operators
- Other Operators
  - Side effect operators
  - Cast operators
- Integer Arithmetic
- Booleans

3

© CRS Enterprises Ltd 2001-15

Operators

- **Binary**
  - **+** add
  - **-** subtract
  - ***** multiply
  - **/** divide
  - **%** remainder
- **Logical**
  - **&&** AND
  - **||** OR
  - **!** NOT
- **Bitwise**
  - **&** AND
  - **|** OR

```
x + y;
x - y;
x * y;
47 / 10;
47 % 10;
```

```
4
7
```

```
if(x > 3 && y < 7) ...
if(!(x <= 3) || (y >= 7) ...
```


The division operator behave differently for integral and floating types. For floating types, division is exact (at least to the precision of the underlying type). However, if both data items are integral, then integer division is performed and the remainder is discarded.

If the remainder is important then it can be retrieve using the % operator. The remainder operator can only be used with integral operands.

There are two types of AND operation. Logical && is the Boolean AND; Bitwise && works at the binary level and is used for low level operations. Similarly considerations apply for || and the NOT operators.

Assignment

```
int main(void)
{
    int a = 10, b = 20, c = 30, d = 40;

    a++;
    a += 1;
    a = a + 1;

    d = d - b;
    d -= b;

    c = c * (b + 10);
    c *= b + 10;

    return 0;
}
```

 29

Rather confusingly, there are 3 different ways to add one to a variable. Historically, these operators generated different code, but with modern optimizing compilers this is no longer the case. Choose the form that you consider the most readable.

The increment and decrement operators are defined as

++	increment by 1
--	decrement by 1

Other operators are as follows:

+=	update by addition
-=	update by subtraction
*=	update by multiplication
/=	update by division
%=	update by modulus (ints only)
|=	update by BIT-ORing
&=	update by BIT-ANDing
^=	update by BIT-XORing
<<=	update by BIT SHIFTing left
>>=	update by BIT SHIFTing right

Side Effects of ++ and --

- **Prefix and Postfix Operators**
  - **increment is a side effect**

```
#include <stdio.h>
int main(void)
{
    int i, x;
    i = 100;
    x = ++i;
}
```

```
#include <stdio.h>
int main(void)
{
    int i,x;
    i = 100;
    x = i++;
}
```

- **= i++** **assignment first, then the side effect**
- **= ++i** **side effect first, then the assignment**

The prefix and postfix operators are called side effect operators. The differences between the operators only becomes apparent when the operators are used within an expression.

If these operators are used in prefix mode then the effect of the operator is immediate; the operand is updated before any assignment.

In postfix mode the reverse is true; the operand is updated after the assignment.

Note:

The order of evaluation of multiple side effect operators on the same operand is not defined by the C standard. This can give rise to ambiguous code.

Exercise 1 - using ++ and --

- **What value is the value of a after each line executes?**

```
#include <stdio.h>

int main(void)
{
      int a = 0, b = 10, c = 1;

      a = ++b  +  ++c;
      a = b++  +  c++;
      a = ++b  +  c++;
      a = b--  +  --c;
}
```


Try this out in a debugger to check your results.

Casting

- **Type conversions (casts) allow you to assign different types at run time**
  - **with possible loss of precision**

```
#include <math.h>
int main(void)
{
    double pi = 4 * atan(1.0);
    float  x = 3455.36263F;
    long   y = 21458904L;

    x = (float) d;
    i = (int) d;
}
```


Casting is where data of one type is being assigned to a variable of another type. The compiler will allow casts between compatible types. Note that casting may loose precision (e.g. double to float).

Note:

The programmer has some control when dealing with constants. The letters L and F can be used to indicate the desire to treat the constant as a long (or long double) or a float respectively.

Arithmetic Involving Different Types

- **Compiler will promote types (automatic cast) where appropriate**
  - **but only in a sub expressions**
  - **watch out for truncation in integer division**
  - **watch out for loss of precision in assignment**

```
int main(void)
{
    int    x = 10;
    double y = 7.77;
    int    result;
    ...
    result = 15 / x + y;
}
```


The C compiler will perform automatic casts in sub-expressions involving different types. The automatic conversion is not sensitive to the context and depends only on the operands in the sub-expression.

This leads to unexpected results in expressions such as

15 / x + y;

This is evaluated as two sub-expressions. In the first sub-expression

15 / x

both operands are integral and the compiler does not cast either operand. This leads to integer division and a result of 1.

The second sub-expression is then

1 + y

and this time the operands are of different types. The compiler will promote the first operand to the double 1.0000. Obviously the decimal part has already been discarded and the final result is 8.77 and not 9.27 as you might have first thought.

Relational Operators

- **Binary operators**
  - **>** greater than
  - **<** less than
  - **==** equal to
  - **!=** not equal
  - **>=** greater than or equal
  - **<=** less than or equal

```
int main(void)
{
    int   x = 100;
    int   y = 200;

    if(x == y) {
        ...
    };

    if(x != y) {
        ...
    }
}
```

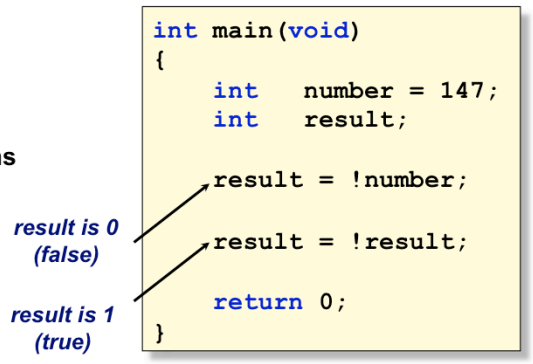

All six relational operators exist in C. Care must be taken with the syntax of the equality and non-equality symbols, since neither != nor == are common in other languages.

Operands should all be of integer or compatible type. Automatic conversions will take place on non integral types. The Boolean value generated by these operators are either the integer 1 (true) or the integer 0 (false).

Be careful to avoid comparison involving floating point numbers, as these are not represented exactly in memory and small rounding errors can cause unexpected results (especially ==).

Booleans

- C89 did not have a Boolean type
  - integers used instead
  - Zero is false
  - Non-zero is true
- You can use #define
  - to define your own Booleans




```
#define TRUE 1
#define FALSE 0
```


C89 didn't have a Boolean type, so integers were used for that purpose. C98 does have a Boolean type, but old habits die hard - you are very likely to encounter the old style.

Any non-zero integer is treated as TRUE by the compiler (e.g. number = 147); zero is treated as FALSE. The relational operators are guaranteed to yield 0 or 1, but are capable of interpreting ALL integral values as Booleans.

In attempt to make C code more readable, many programmers define their own Booleans using the preprocessor:

#define TRUE 1

#define FALSE 0

&& Operator

- **logical AND operator**
  - false && false = false
  - false && true = false
  - true && false = false
  - true && true = true
- **short-circuit evaluation**
  - expressions evaluated from left t
  - evaluation stops as soon as resu

```
int main(void)
{
    int a = 42;
    int b = 0;
    int result;

    if(a > 50 && 15/b > 1) {
        ...
    }
}
```

divide by zero doesn't happen

 36

Unlike the majority of C operators, the AND and OR operators are evaluated in strict order. Both perform evaluation from left to right. If the truth value of the expression is determined on performing the left operand evaluation, further evaluation does not ensue. This is called short-circuit evaluation and is important in conditions like

a > b && 15/(a -b)

where the second expression could potentially cause a divide by zero error. The short circuit evaluation ensures that this never happens, because if a - b does equal zero, the first condition will fail and the short circuit will return FALSE immediately.

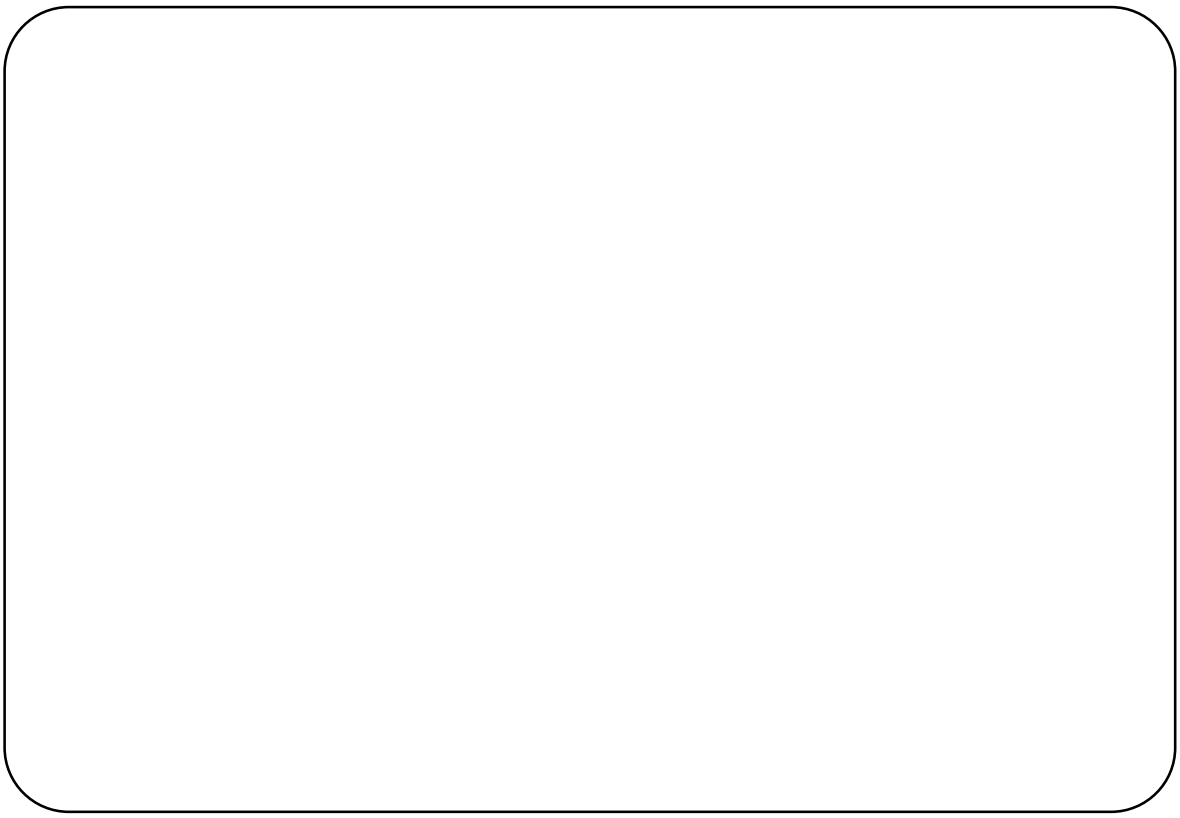
Care must be taken to use the repeated symbols && and ||. The symbols & and | on their own are used for bit manipulation.

|| - the OR Operator

- logical OR operator
  - false || false = false
  - false || true = true
  - true || false = true
  - true || true = true
- short-circuit evaluation
  - as with &&

```
int main(void)
{
    int x = 0;
    int y = 22;

    if(x > 10 || y < 30) {
        ...
    }
}
```


Similar considerations apply to the OR operator.



Pointers

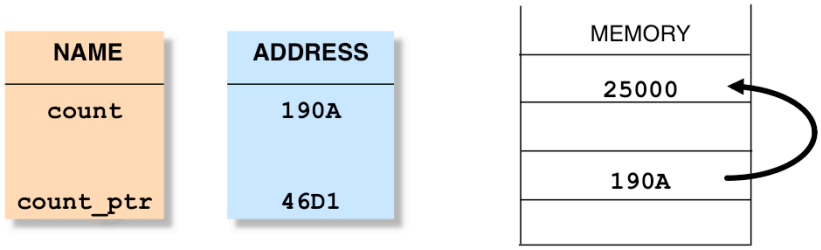
- **Pointers**
  - the concept
  - uses of pointers
  - declaring pointers
- **Operators**
  - & operator
  - * operator
- **Complex Pointers**
  - pointers to pointers

4

© CRS Enterprises Ltd 2001-15

Indirection

- A pointer is a variable that contains the address of another variable
- A pointer contains the address of a variable
  - allowing access to the variable indirectly



A pointer is a scalar data item that contains the address of some other variable. In the example above a variable count has the value 25000 and is stored at address 0x190A. The pointer count_ptr has the value 0x190A which is the address of count. The address of count_ptr is unimportant.

Pointers provide an indirect method of accessing data in C. Applications typically make extensive use of pointers and are an essential tool in the C programmer's repertoire.

Pointers

- **Pointers allow us to...**
  - **allocate memory dynamically**
  - **change values passed as arguments to functions**
    - **call by reference**
  - **deal with arrays concisely and efficiently**
  - **effect whole-structure operations**
  - **represent complex data structures**
    - **such as linked lists, trees, stacks**
- **Pointers are essential for writing efficient programs**
  - **but common source of buggy code**
  - **uninitilised pointers can crash programs**

 41

It is essential to get to grips with the concept of pointers. It is the only way that some techniques can be achieved.

Some of the use of pointers include:

allocation of arbitrary amounts of memory at run time.

call by reference semantics when working with functions.

deal with arrays concisely and efficiently

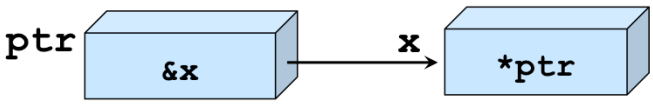
effect whole-structure operations

represent complex data structures such as linked lists, trees and stacks

The downside of using pointers is that they tend to result in code that is difficult to read and maintain and are often the source of obscure and intermittent bugs.

Declaring Pointers

```
int* int_ptr;      // pointer to int
long* lp;          // pointer to long
double* dp;        // pointer to double
int i,j,k,*ip;     // i, j and k are integers
                   // *ip is a pointer to int
long** q = &lp;    // pointer to pointer to long
```

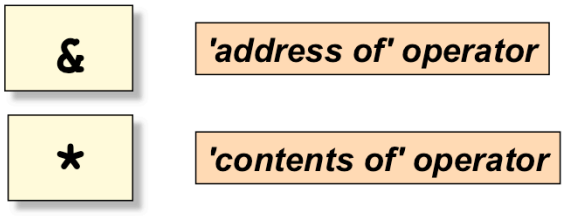


An individual pointer is constrained to contain addresses of only one type. The type is specified in the declaration. Therefore, a pointer is declared as a 'pointer to type'. The position of the * in the declaration is important, but any amount of white space can be used. It is recommended that the * should be written next to the pointer's type and that pointers and non-pointers should not be declared in the same statement. If you do mix pointer and non-pointers in the same declaration it will not be possible to place the * next to the pointer's type (see the fourth declaration above).

Note that the last example includes an initialization. The & operator is used to generate the address of its operand. The * operator is used to dereference the pointer (retrieve the data at the end of the pointer).

Pointer Operators

- **Two unary operators used with pointers**



The & operator generates the address of a previously-declared variable or const.

The * operator accesses the value contained at the address contained in the pointer.

These two operators are inverse operators.

Declaring Pointers

-

```
int* px;
```

- *px can be thought of as an int
  - may be used anywhere an integer variable can be used

```
int main(void)
{
    int x = 10;
    int y;
    int* px = &x;  // initialise px to point to x

    x = *px + 1;   // increment x
    y = *px / 2  + 10  - 7;
    if(*px > 10)
        printf("*px is %i\n", *px);
}
```

A pointer is bound to a particular type

To declare a pointer to an int use

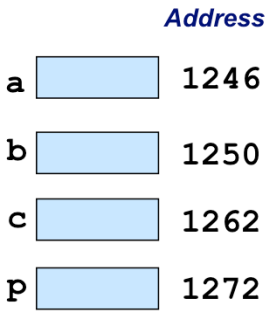
int* px

The declaration restricts the pointer such that it can only ever point to an int. To dereference the pointer use *px. Thus you can think of *px being an alias for the variable at the end of the pointer. The alias *px can be used wherever it is legal use an int.

Using & and *

- **Fill in the values of a, b, c and p**

```
int main(void)
{
     int a = 46;
     int b = 19;
     int c;
     int* p = &a;

     c = *p;
     p = &b;
     c = *p;
     b = 77;
     c = *p;
}
```

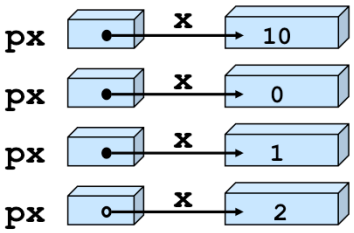


The actual addresses used in this example are not important. Work through this example in a debugger to check out what happens to b, c and p.

Manipulating Pointers

- **Pointer dereferences can occur on the left-hand-side of assignments**

```
int main(void)
{
   int x = 10;
   int* px = &x;

   *px = 0;
   *px += 1;
   (*px)++;

   return 0;
}
```



Since *px can be thought of as an integer variable, you can assign to it! In the above example px points at x and therefore *px is an alias for x. The three statements involving *px are equivalent to

x = 0;

x += 1;

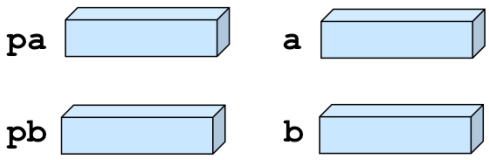
x++;

Note the parenthesis in the last example.

Data and Pointers

- **Distinguish between**
  - ***p (an integer) and p (a pointer)**

```
int main(void)
{
   int a = 7;
   int b = 3;
   int* pa = &a;
   int* pb = &b;

   *pa = *pb;
   pa = pb;
}
```



Pointers are data items whose values are addresses. If a pointer is accessed for manipulation, it will be an address that is being processed. Once a pointer has been dereferenced using the * operator, it is the data pointed at by the pointer that is being processed. The compiler will warn you about mismatches such as *pa = pb.

The statement

*pa = *pb

is one of integer assignment, as both *pa and *pb are ints. The effect is to assign 3 to a.

However, the statement

pa = pb

is one of pointer to integer assignment, as both pa and pb are pointers to ints. The effect is to assign b's address to pa, i.e. both pointers now point to b.

* operator

- The * operator has different meanings in different contexts
  - often leads to confusing code

```
int main(void)
{
    int i = 3, j = 2, k;
    int* p = &i;
    int* q = &j;

    k = *p**q;
    k = (*p)++**q;
    k = *p/*q;
}
```

 48

Beware that the * operator has different meanings in different contexts and this often leads to confusing code. Consider

k = *p**q; this means k equals *p multiplied by *q

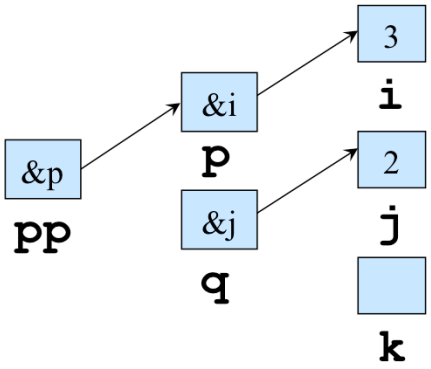
k = (*p)++**q; this means k equals *p multiplied by *q and *p is incremented

k = *p/*q; this doesn't even compile - the intention is to assign *p divided by *q to k, but /* is the start of comment sign

Pointers to Pointers

- declare as int**

```
int main(void)
{
    int i = 3, j = 2, k;
    int* p  = &i;
    int* q  = &j;
    int** pp;

    pp = &p;
    printf("%i", **pp);
    p = q;
    printf("%i", **pp);

    k = *p * **pp;
}
```

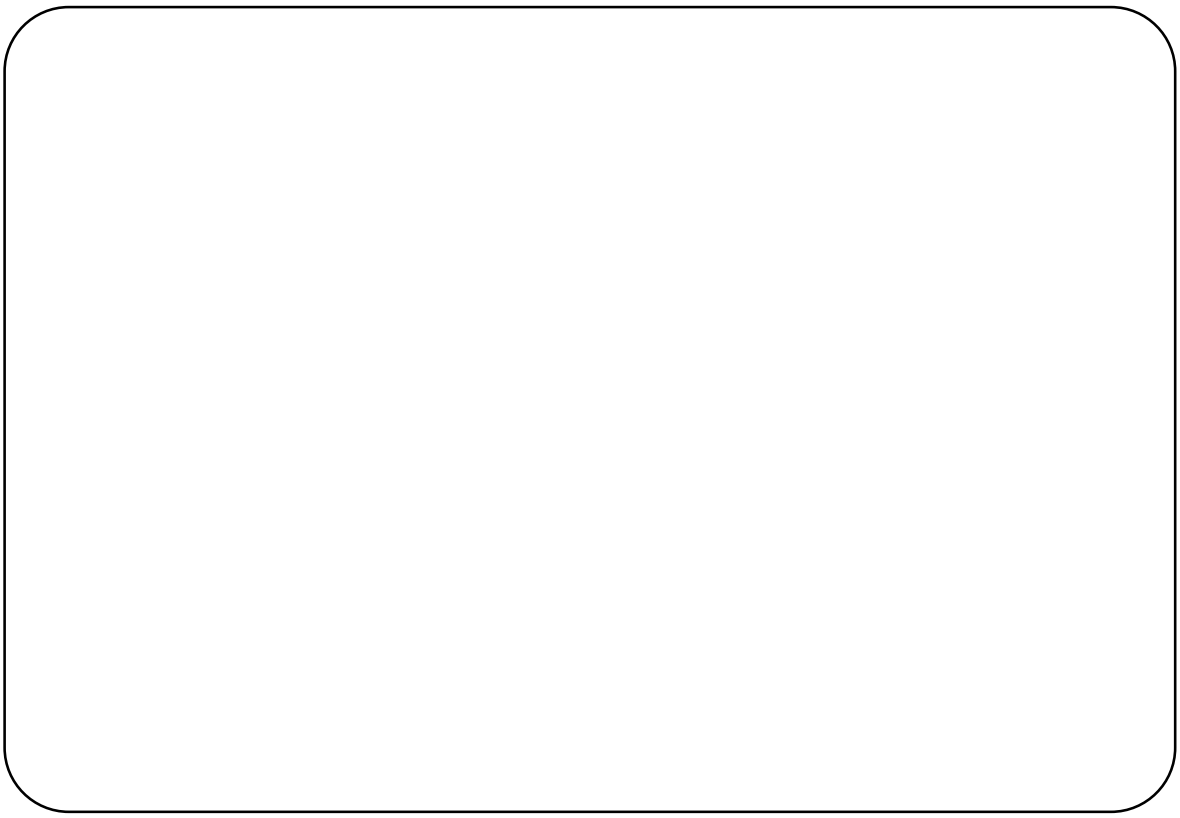


Pointers can be used to point to other pointers. The declaration int** pp should be read as pp is a pointer to an int* (itself a pointer to an int). As with other pointers the runtime expression **pp is an alias to the int at the end of the pointer chain.

Note the last expression

k = *p * **pp

*p is an alias for the data pointed at by p and **pp is an alias for the data pointed at by the pointer chain starting with pp. These two integers are multiplied together and stored in k. Try using a debugger to check what value gets assigned to k.



If Statements and Loops

- **Conditionals**
  - if statement
  - switch statement
  - conditional operator
- **Iteration**
  - while statement
  - do while statement
  - for statement

5

if Statement

```
if (expression)
    statement
```

```
if (d > 0.0) printf("d is positive\n");
```

```
if (d > 0.0)
{
    printf("d is positive");
    printf("\n");
}
```


The simplest form of the if statement is shown above. The conditional expression must be enclosed with parentheses and is followed by the body. The body can be either a single statement or a block of statements.

if - else Statement

```
if (expression)
    statement
else
    statement
```

```
if (d > 0.0)
    printf("d is positive\n");
else
    printf("d is not positive\n");
```

```
if (d > 0.0){
    printf("d is positive");
    printf("\n");
} else {
    printf("d is not positive");
    printf("\n");
}
```


If statements can have an else branch. The body of the else branch can be a single statement or a block of statements. It is perfectly permissible to mix a single statement in the if branch with a block in the else branch or vice-versa.

Thus there are 4 forms of the if-else statement:

if(condition) statement; else statement;

if(condition) statement; else block-of-statements;

if(condition) block-of-statements; else statement;

if(condition) block-of-statements; else block-of-statements;

Multiple if Statements

```
if (expression)
    statement
else if (expression)
    statement
...
else
    statement
```

```
#include <stdio.h>

int main(void)
{
    int legs = 6;

    if (legs == 1)
        printf("Flamingo\n");
    else if (legs == 2)
        printf("Ape\n");
    else if (legs == 4)
        printf("Dog\n");
    else
        printf("Insect\n");
}
```

- **This form allows for 'Multi-way decision making'**

By combining several if statements you can create multi-way decision logic. Note that this is not new syntax; it is simply multiple application of the if statement. If you choose the layout shown above, the multi-way decision becomes very readable.

The switch Statement

- **Multiway selections**
  - expression determines case
  - break to exit the switch
  - drop through if break isn't used
- **Case index must be integral**
  - or char
  - can't use strings for cases

```
switch(legs)
{
case 1:
    printf("Flamingo\n");
    break;
case 2:
    printf("Ape\n");
    break;
case 4:
    printf("Dog\n");
    break;
default:
    printf("Insect\n");
    break;
}
```


Tests based on the evaluation of a simple integral expression can be expressed using the switch statement. The switch is neater and more efficient than a corresponding nested if/else construct. Once the syntax is in place, the flow is relatively easy to read. The rules are few but important and are detailed on the next page.

Each case can contain multiple statements and it is not necessary to enclose these statements in {}. The break statement transfers control to the end of the switch. Without the break, control simply passes to the next case, and although perfectly legal, probably not what was intended (but see next page for a valid example of omitting the break).

Note that the switch only works with integral types.

No break in switch Statement

```
int verse ;
for (verse = 1; verse != 5; ++verse)
{
     printf("On the ");
     switch (verse)
     {
          case 1  : printf("1st");          break;
          case 2  : printf("2nd");          break;
          case 3  : printf("3rd");          break;
          default : printf("%dth", verse);  break;
     }
     printf(" day of Christmas my true love sent to me:\n");
     switch (verse)
     {
          case 5 : printf("five gold rings, ");
          case 4 : printf("four calling birds,  ");
          case 3 : printf("three french hens, ");
          case 2 : printf("two turtle doves, and  ");
          case 1 : printf("a partridge in a pear tree\n");
     }
}
```


In this program the break statements have been omitted in the second switch statement to give a drop through effect. As you can readily see, this can be useful occasionally. A more common usage for the drop through would be:

```
case 3:
case 4;
case 5;
     // statements for all three cases
```


The ?: Conditional Operator

- 3 operands
  - expression1 ? expression2 : expression3

```
max = (x > y) ? x : y;
min = (x > y) ? x : y;
```

- More concise than if statement
  - but more cryptic?

```
if (x > y)
    max = x;
else
    max = y;
```


The ternary conditional expression operator is an efficient alternative to a simple if/ else construct. Care must be taken to ensure type matching and precedence.

The conditional expression operator was used extensively to optimize code before the advent of optimizing compilers - nowadays its use is a style choice.

The while Statement

- **The simplest iterative statement**

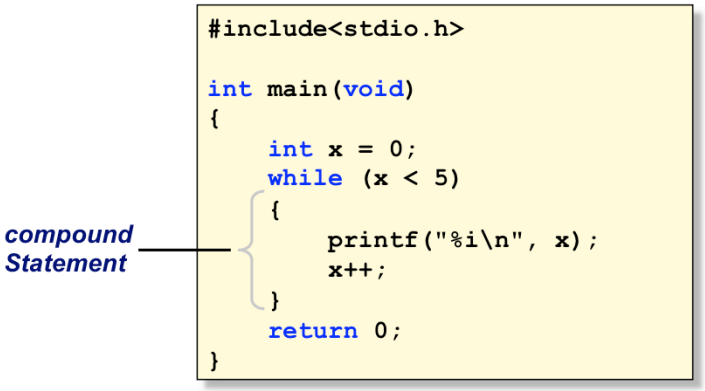

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    while (x < 5)
        printf("%d\n", x++);
    return 0;
}
```

 58

The while loop consists of a test on entry, followed by a statement. This statement can be a simple statement (as shown above) or a block of statements enclosed in {}. The statement/block is executed only if the test evaluates to true. Note that the if the test fails on the first iteration, the block will never be executed.

Compound Statement

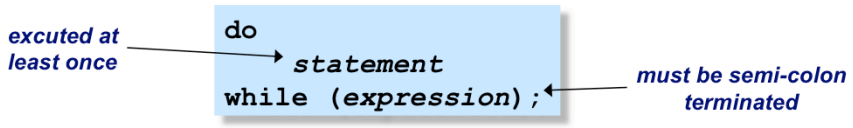
- **The statement part of the loop can be a simple statement or a compound statement**



The example above shows how to use a block as the body of loop. The block is called a compound statement. Normally a block consists of two or more statements, although it is legal to place a single statement (or even no statements) in a block.

The do while Statement

- **Test is at the end of loop**



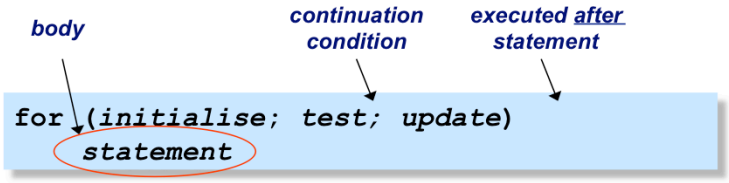

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    do
        printf("%d\n", x++);
    while (x < 5);
}
```


The do while loop is the inverse of the while statement in the sense that the test is at the end of the body. The body is repeatedly executed while the test evaluates to true. This form of loop implies the body is executed at least once.

Note the syntax. The do and the while enclose a block of statements. As with the switch statement, the block does not have to be enclosed in {}. A common error is to forget to enclose the test in parentheses.

The for Statement

- **The generic iterative statement**




```
#include <stdio.h>

int main(void)
{
    int x = 0;
    ...
    for (x = 0; x < 5; x++)
        printf("%d\n", x);
}
```

 61

The for loop is an alternative to the while statement, with separate clauses for initialization, test, and update. The body can be a single statement or block of statements. Note that the initialization, test, and update clauses are separated by semi-colons.

The order of the three clauses and body is as follows:

initialise

is performed once only and then

test

body

update

are repeated until the test fails.

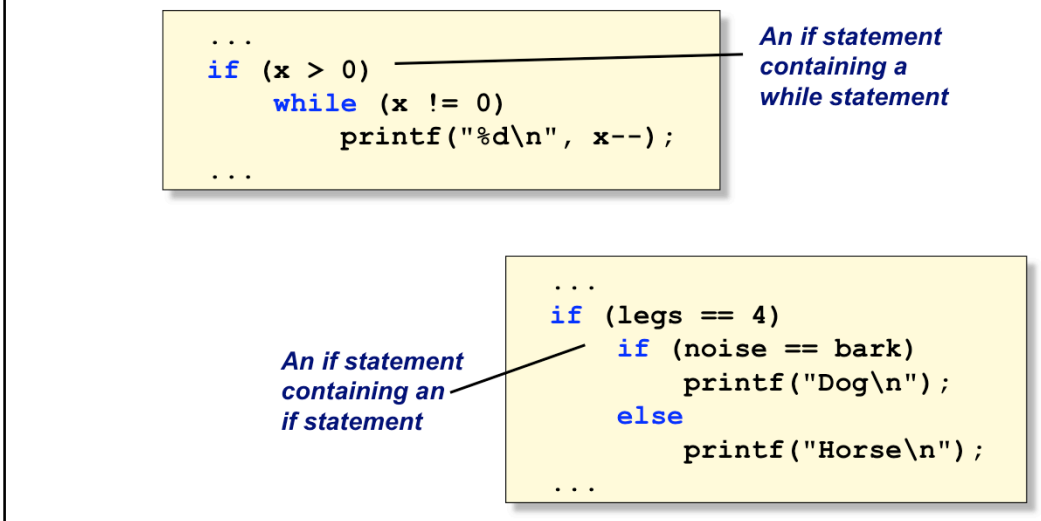
Note that the initialise and update clauses can be blank or can contain multiple comma separated expressions:

for(; x < 5; x++) { ... }

for(a = 0, b = 0; a + b < 5; a++, b++) { ... }

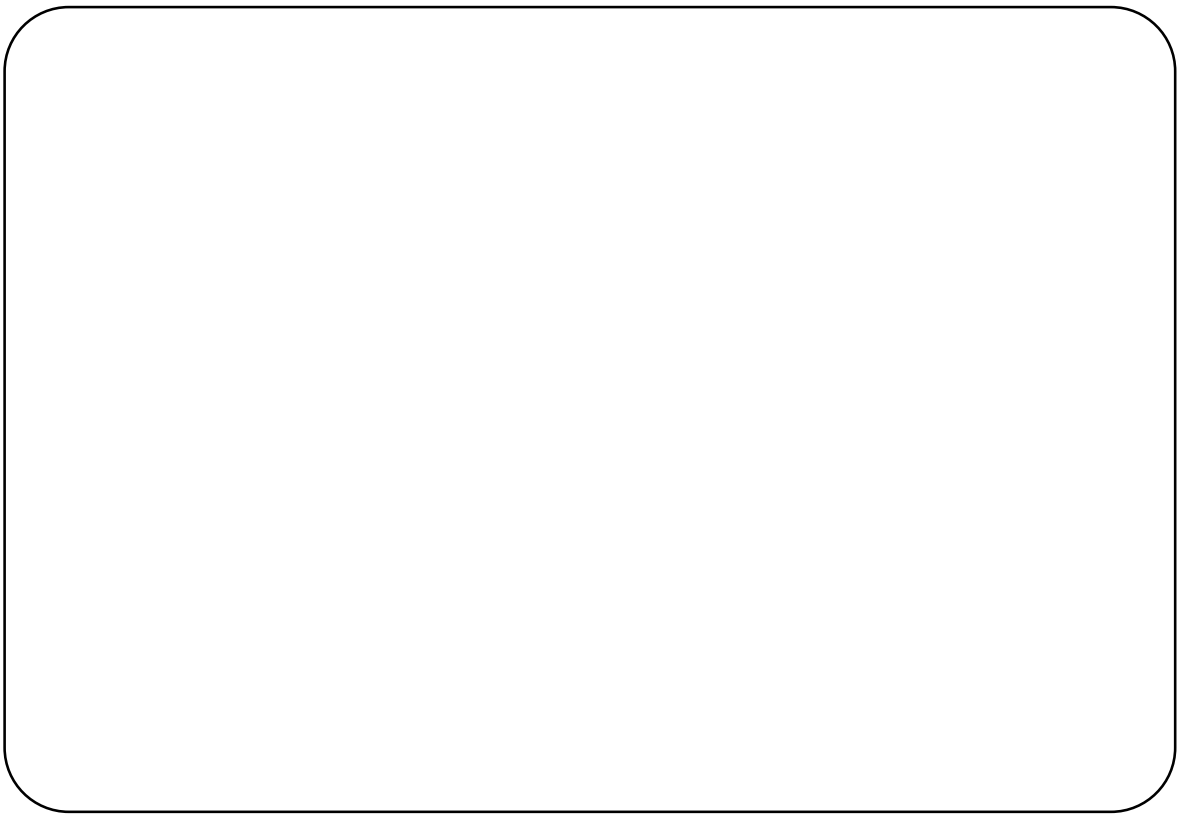
The test can also be omitted, in which case we get an infinite loop. In such cases a

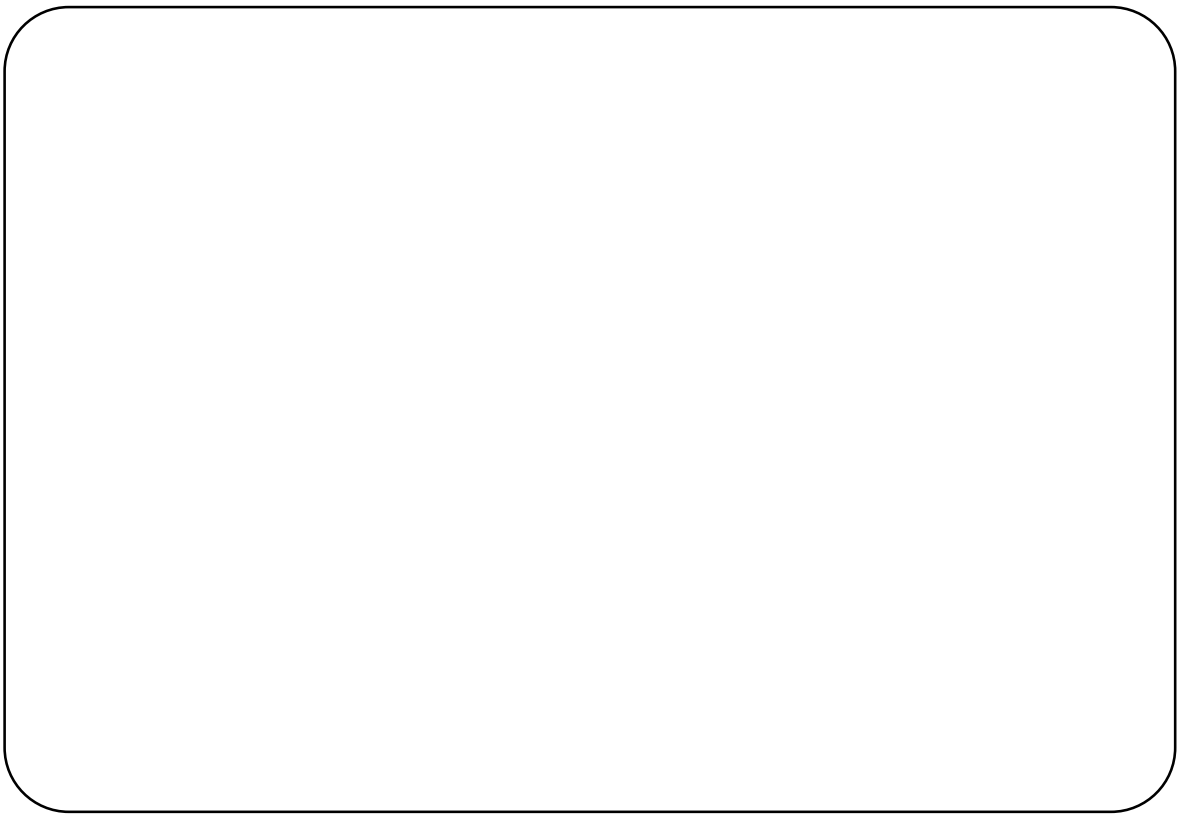
Nested Control Flow



The body of an if or while can be a simple statement. This statement can itself be an if or while statement. If this statement spans several lines readability suffers. Many coding styles insist on the presence of braces for all these constructs to make the extent of the body obvious, even when there is only one statement in the body. The first example could be rewritten as:

```
if (x > 0)
{
   while (x != 0)
   {
       printf("%d\n", x--);
   }
}
```



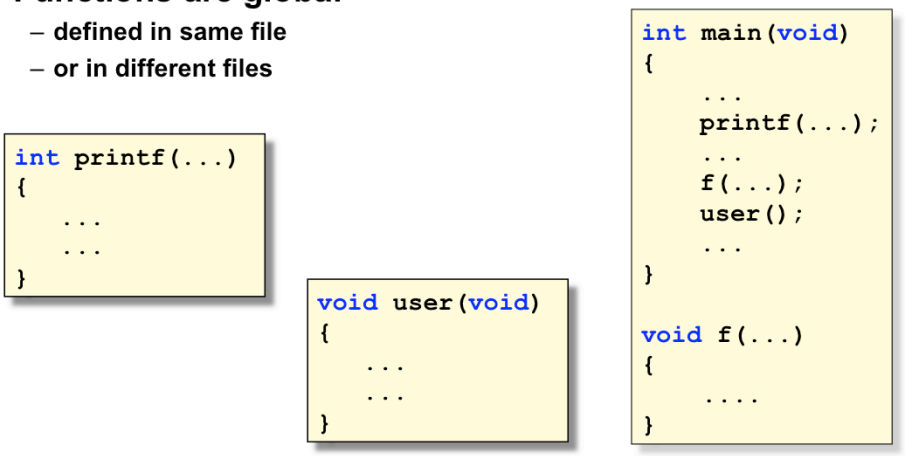
Functions

- **Program Structure**
  - main
  - prototypes
- **Parameters**
  - pass by value
  - pass by reference
  - return statement
- **Data**
  - local and global data
  - static data
  - register data

6

C Program Structure

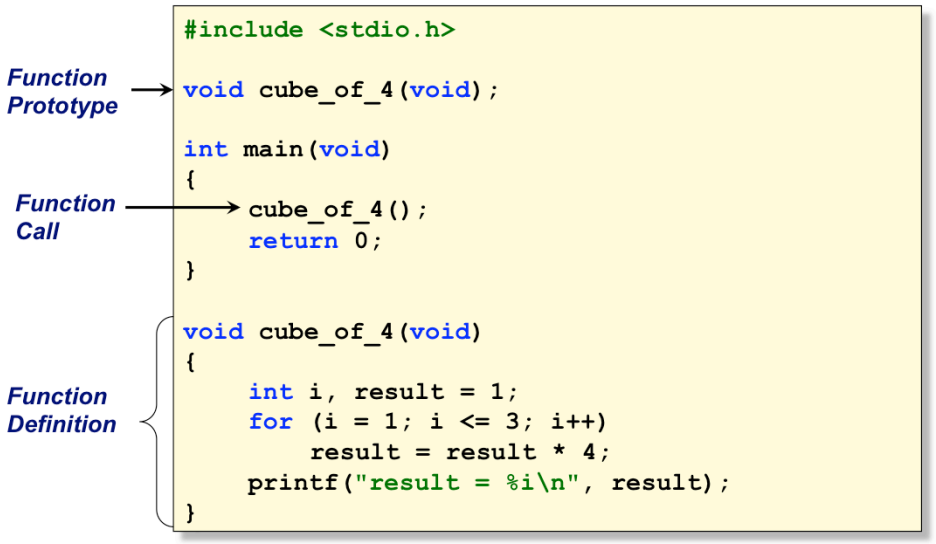
- **Program starts at main**
- **Functions are global**
  - **defined in same file**
  - **or in different files**



Every C program consists of a set of functions that communicate with each other. The execution of the program begins with main and continues sequentially through main's code statements, which invoke functions as they appear; execution ceases at the end of main.

All functions are global in definition, which means they can be called from within the same file or a from an external file. Calling library functions uses exactly the same syntax as calling user defined functions.

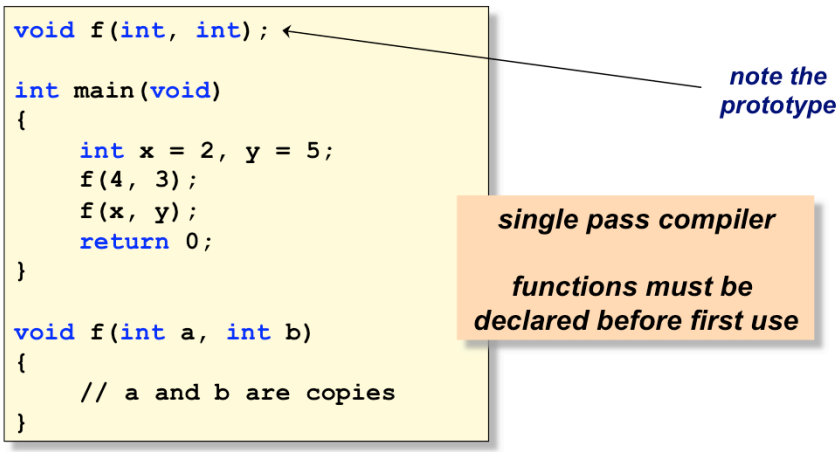
Simple Functions



Simple functions can be defined that have no parameters and return nothing. The C89 standard uses the void keyword both to denote no input parameters and no return value.

Because the C compiler is single pass compiler, you must declare a function before its first call. To avoid having to define main as the last function, function prototypes are used as forward references to functions. Function prototypes allow the compiler to check the number and type of input parameters and the return type on any call to the function. By using function prototypes, we can put the actual function definition in a separate file (although we haven't done that in the example above).

Passing Parameters



When we call a function in C, we pass copies of the parameters specified in the call (call by value). In the example above the function f is called and copies of the constants 4 and 3 are passed in the first call; copies of the variables x and y are passed in the second call.

Note that the function definition defines the two input parameters as a and b. However the prototype is only used to check that two ints are passed as parameters and therefore it is not necessary to specify variable names in the prototype. Hence the prototype can be written with or without variable names:

void f(int x, int y);

void f(int, int);

The return Statement

- **Return statement**
  - **ends execution of the statements in a function**
  - **returns control to the calling function**

```
void do_it(void) ;

int main(void)
{
    ...
    do_it();
    ...
    do_it();
    ...
    return 0;
}
```

```
void do_it(void)
{
    ...
    ...
    if (...)
        return;
    ...
    ...
}
```


The return statement sends control back to the statement that invoked the function. The return statement is usually the last statement in the function, but it is permissible to place a return anywhere (as in the second example).

The return value must be an expression that is the same type as defined by the function (void represents no return type). For example, main returns an int which is used as the process exit code.

With void functions, the return statement is not mandatory. Non void functions can have as many return statements as desired.

Returning Values

- The general form of the return statement is:

```
return expression;
```

- Using the return statement, functions can pass a single value back to the calling function

```
double average(double, double);     /* prototype */

double average(double x, double y) /* definition */
{
    return (x + y)/2.0;
}
```


Output information is defined in the prototype in the same way as input information (using a type). However, only one item can be returned. If you wish to return multiple items, you will have to wrap them up in a single struct (see later).

Function Prototypes

- Function prototypes have the following form...
-

```
return_type  name ( parameter_list ) ;
```

- If parameter_list is omitted, no argument type checking is performed and default argument conversions occur!

```
int f1();
int g1(void);
```

use void to indicate 'no parameters required'

- If return_type is omitted, int is assumed

```
/*int*/ f2(void);
void g2(void);
```

use void to indicate 'no value returned'

Function prototypes should always be used. If you omit them, then the compiler will not be able to check that the correct parameters are passed or returned.

If you omit the parameter types in the call, then the compiler will assume you have passed the correct types to the function. For example, a function expecting two doubles will fail if you pass two ints unless you also supply the prototype. Without the prototype, the compiler will get confused. If you supply a prototype, automatic conversions will take place. Similar considerations apply to return values.

Pass by Value

why does the swap function fail?

```
void swap(int, int);
...
int main(void)
{
    int x = 100, y = 200;
    swap(x, y);
    return 0;
}

void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```


C always passes parameters as copies (pass by value). In this example, swap actually swaps the copies of x and y and not x and y themselves. Hence, this swap function does not work.

If you really want to swap x and y you must use pointers (see later).

Pass by Reference

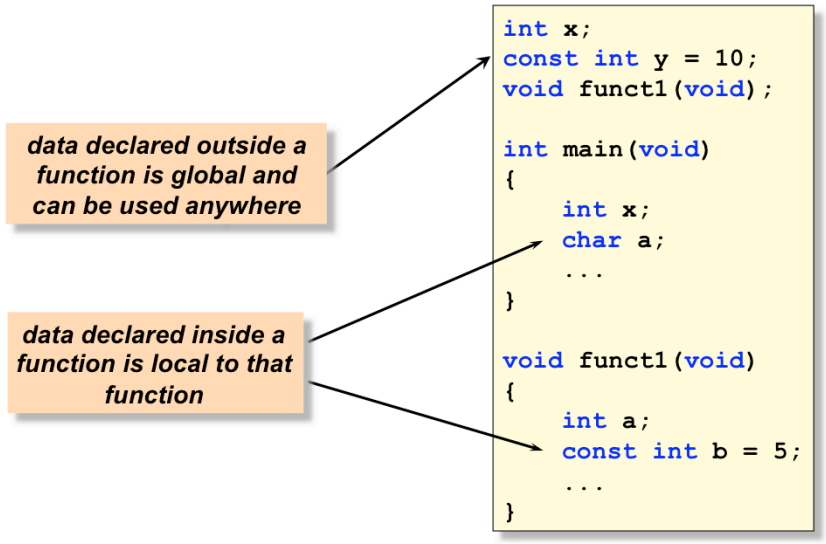
- use pointers
  - to modify original data
- use pointers on the call
  - swap(&x, &y)
- use pointers in the function
  - void swap(int*,int*)

```
void swap(int*, int*);
...
int main(void)
{
    int x = 100, y = 200;
    swap(&x, &y);
    return 0;
}

void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```


Although C compilers do not support call by reference directly, you can simulate call by reference using pointers. For the swap function, simply pass the address of the variables you want to swap (i.e. pointers to these variables) and then the function has indirect access to these variables. In the above example, *a is effectively an alias for x and *b is effectively an alias for y.

This is the standard way to write a swap function in C.

Local and Global Data



 74

Scope refers to the accessibility of data. There are two types of scope: local and global.

Local scope defines access constrained to a block. Data, whether variable or constant, defined inside any block has local scope. The inference is that only statements within the block are able to access that data directly by name.

Global scope defines a wider scope. Data declared outside a function block is potentially accessible to the entire program. By default, all statements in the source file below the declaration have access.

If a local variable shares its name with a global variable, the global variable masks the local variable.

It may be worthwhile using your own naming conventions to distinguish between local and global identifiers.

static Data

- **static data items...**
  - **created on first entry to the function in wh**
  - **retain their values through successive fur**
  - **default initialization to zero**

```
int add_to_total(int c)
{
  static int total;

  total += c;
  return total;
}
```

variables declared outside a function have static storage by default

```
/*static*/ int count;

int main(void)
{
  ...
}
```


The static storage class describes the property of data that exists throughout the lifetime of the program execution. There are two types of static data: global data and static local data.

The local data needs to be designated by using the static keyword in the declaration. A local static always exists. The compiler allocates special ‘static’ storage for the data before the program is executed. Unlike normal local variables, static locals remain in memory between calls.

Global data is static by default and does not require the static qualifier. In fact, using the static keyword on global data is used for a different purpose: to define the scope of the global (i.e. to make it visible across multiple compilation modules).

Because the compiler is responsible for allocating storage, the initial value of all static variables is zero unless explicitly initialised. Local statics will receive the initial value once and once only.

Storage Class register

- The register variable prefix advises the compiler of heavy usage
  - hint to use a machine register
  - may de-optimise the optimiser
- Not to be used with modern optimizing compilers

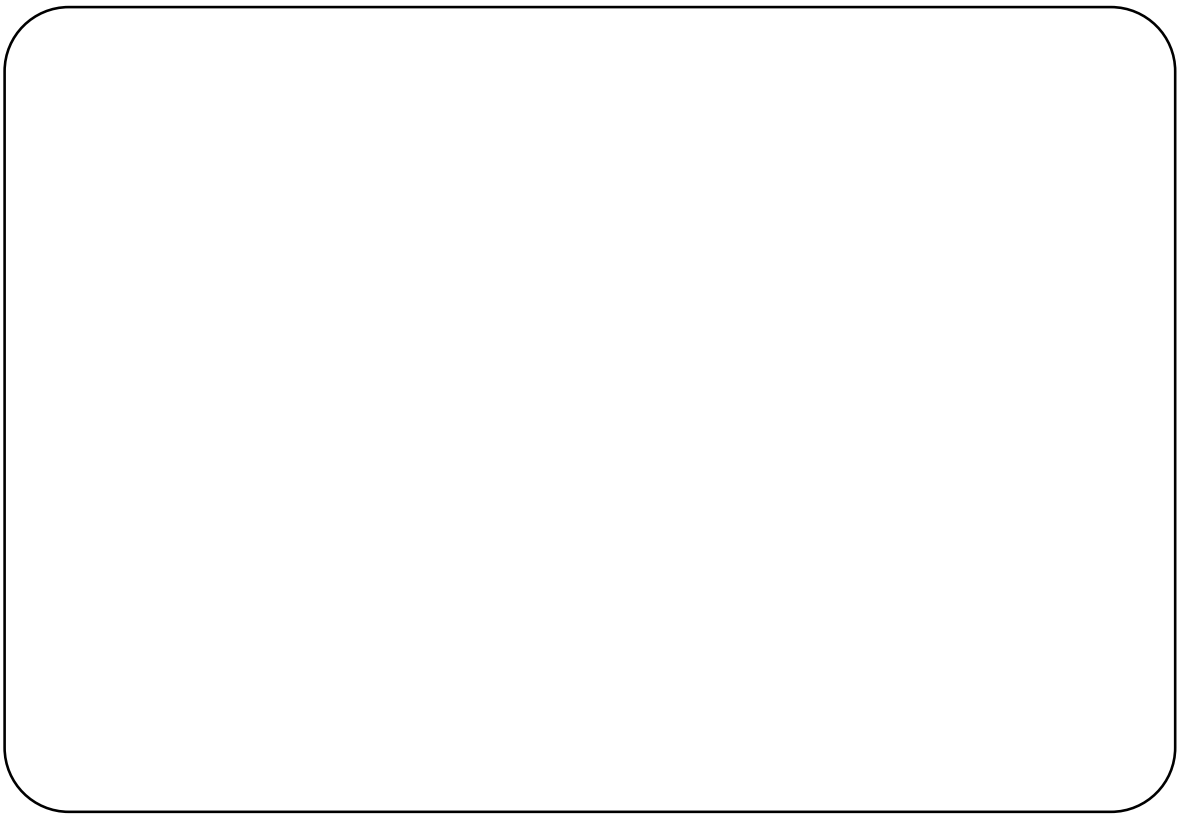
```
int parameter(register int used_lots)
{
    ...
    ...
}
```

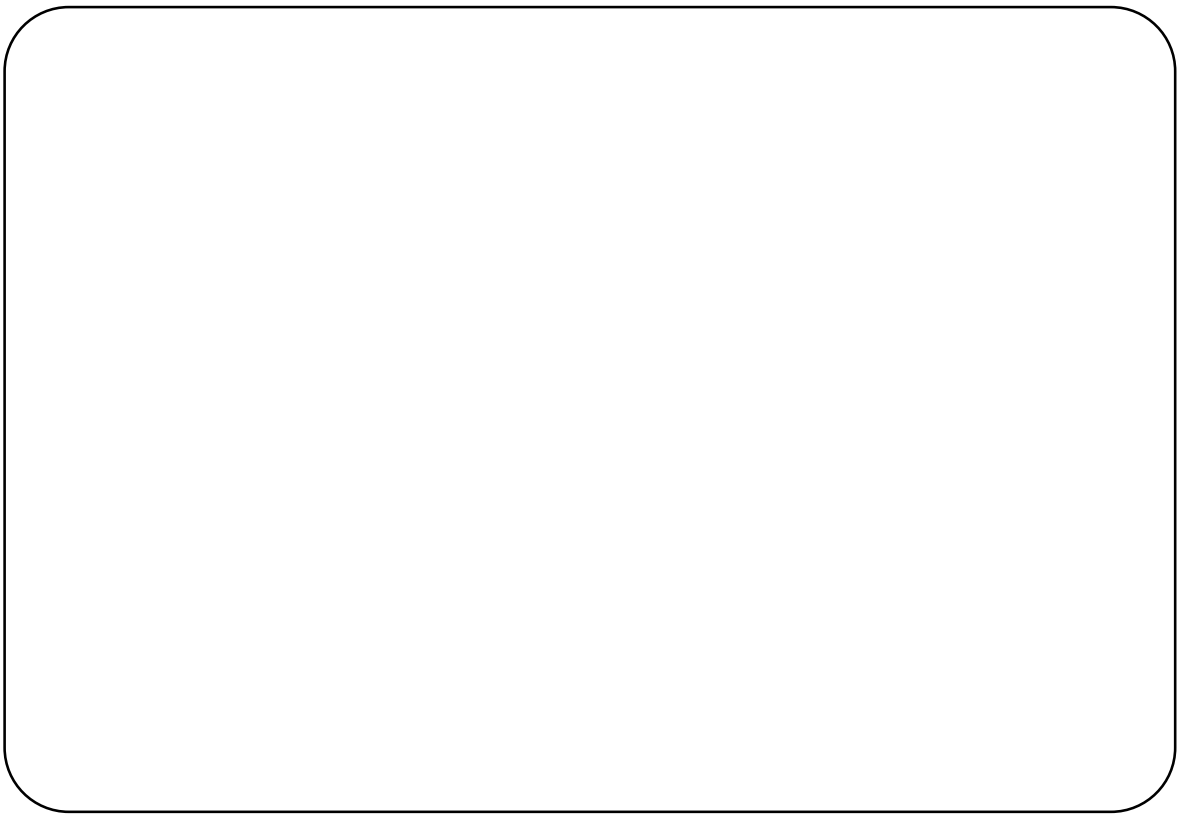
```
int counter(int x)
{
    register int master_count;
    ...
}
```


'Register' variables have to be local variables or function parameters. If no register is available, the data is stored as normal local variable.

However, nowadays compilers are better placed than most programmers at deciding how CPU registers should be used. Fine-tuning using this technique is best left to the compiler. Using the register keyword is now effectively an anachronism.

Use the volatile keyword on a variable to stop the optimizing compiler using register variables. It tells the compiler that the item could potentially be modified by another process (e.g. through inter-process communication such as shared memory) and that the compiler must not optimize.





Arrays

- Arrays
  - declaring
  - initialising
  - accessing
- Functions
  - passing arrays as parameters
  - library functions
- Advanced Arrays
  - arrays of characters
  - multi-dimensional arrays

7

Declaring Arrays

- General form of an array declaration

```
type name[size];
```

- The size specifies the number of elements in the array. Each element is of the specified type

```
int numbers[10];

int main(void)
{
    double vector[100];
    char   line[132];
    ...
}
```

declares numbers as an array of 10 ints

declares vector as an array of 100 doubles

declares line as an array of 132 chars

- Array elements are stored in contiguous memory

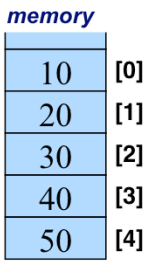
Array are declared with a fixed size and are strongly typed (they can only contain a single type). arrays can be declared as global or local variables.

Note that it is not possible to change the size of an array at run time. However, it is possible to emulate dynamic arrays using pointers (see a later chapter).

Declaring and Initialising Arrays

- **Arrays may be initialised when declared**

```
int numbers[5] = { 10, 20, 30, 40, 50 };

int main(void)
{
    double vector[100] = { 3.8, 12.6 };
    char name[ ] = { 'G', 'e', 'o', 'r', 'g', 'e', '\0' };
    ...
}
```



Just like scalar data, an array can be initialised at declaration time. The syntax is similar to that of scalar data with the = operator is used. Normally you will give an initial value to each element, but partial initialisations are allowed. The initialisers must be of the correct type, separated by commas and enclosed in { }.

If there are too few initial values, sometimes referred to as a 'short-fall', the remaining values are set to zero. Too many give rise to a compiler error.

On occasions, the size in the declaration can be left for the compiler to fill in. This is achieved by leaving the size blank and by supplying the correct number of initial values. Note that some compilers do not guarantee to set uninitialised elements to zero even though it is part of the C89 standard.

Accessing Array Elements

- **Elements can be accessed using run-time expressions**


```
double dd[3] = { 2.2, 3.3, 4.4 };

int main(void)
{
    double d;
    int x = 10;
    d = dd[0];
    d = dd[x / 5];
    dd[1] = d;
    ...
}
```

no array bounds checking in C

Array indexing always starts at 0. Thus an array of N elements is indexed using the range 0 to N-1. The index expression must be integral.

Unlike languages like Java and C#, the C compiler does not ensure accesses beyond the end of an array are prohibited; we say that the is no array bounds checking in C. This can be the source of very difficult to diagnose bugs and is a major drawback with the design of the language.

Since the compiler does help, it is vital to build bound checking into your code. The onus is on you!

Exercise

- C does not support complete array operations
- Arrays are manipulated element by element

```
int a[10] = { 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

int main(void)

{

    int b[10];
    int i;
    for (i = 0; i < 10; i++)

    {
        // b = 0;
        // b = a;
        // print b
        // input b

    }
}
```

How do we implement these operations?

83

Unlike languages like Fortran, PERL or Python, C does not provide any support for complete array operations such as assignment. Instead, everything has to be achieved through individual element access.

The examples above are implemented using loops:

1) b = 0

```
for (i = 0; i < 10; i++) b[i] = 0;
```

2) b = a

```
for (i = 0; i < 10; i++) b[i] = a[i];
```

3) print a

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

4) input b

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

Arrays and Functions

- **To pass an array to function**
  - **use the name of the array**
  - **arrays passed by reference**

```
void change(double []);
int main(void)
{
      double vector[5] =
          { 0.0, 1.1, 2.2, 3.3, 4.4 };
      change(vector);
      printf("%lf %lf", vector[0], vector[3]);
      return 0;
}
void change(double da[])
{
      da[0] = 3.14159;
      da[3] = da[2] + da[4];
}
```

memory
0.0
1.1
2.2
3.3
4.4

 84

When arrays are passed as parameters to functions only the address of the first element in the array is passed. This address is a pointer to the array. The function receives a copy of the pointer in line with C's call by value semantics. This has several implications.

The address of the first element of the array tells the function nothing about the size of the array being passed. In fact the function has no way of determining the size of the array passed (unless the size is passed as a separate parameter). If you use the sizeof operator it always returns 4 bytes - the size of the pointer being passed!

Since the function can't tell the size of the array, you omit the size in the prototype:

```
void change(double [ ]);
```

Alternatively you can write

```
void change(double*);
```

Since access to the array being passed is through a pointer, the changes to the array inside the function will modify the original array. This is often described as "arrays are passed by reference", but in reality the pointer is passed by value.

Finally since the function has no way of knowing the size of the array being passed, you must be very careful not to write beyond the bounds of the array - best practice is to pass a second parameter defining the size the array.

Example

```
void initialise_array(int [], int);

int main(void)
{
      int x[100];
      int y[7];
      initialise_array(x, sizeof(x));
      initialise_array(y, 7);
      return 0;
}

/* Initialise array to zeros */
void initialise_array(int ai[], int size)
{
      int i;
      for (i = 0; i < size; i++)
            ai[i] = 0;
}
```


The examples above follow the recommended practice of passing the size of the array as a second parameter.

Note that sizeof(x) does return the number of bytes in the array (probably 4 * 100) because x really is an array. If we tried sizof(ai) inside the function it would return 4 bytes, because despite appearances, ai is a pointer and not an array.

Exercise

- **How do we write sum()?**

```
int sum(int[], int);

int main(void)
{
      int theArray[5] = { 1, 4, 9, 16, 25 };
      int total = 0;
      total = sum(theArray, 5);
      printf("Sum = %i\n", total);
      return 0;
}

int sum(int a[], int size)
{


}
```

How do we implement this function?

The sum function can be implemented as follows:

```
int sum(int a[], int size)
{
   int i;
   int sum = 0;
   for(i = 0; i < size; i++)
   {
      sum += a[i];
   }
}
```

Strings - Arrays of char

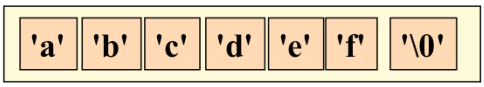
- No language support for strings (no 'string' type)

```
string instrument;
```

- Strings are implemented in C using arrays of char

```
char instrument[64];
```

- The convention is that strings must be terminated by the '\0' character



 87

Strings are implemented in C as arrays of characters. In most languages, strings are variable size, but in C arrays are always fixed in size. In order to introduce some flexibility, the character array defines the maximum size of the string. Then when the string is given a value, a null character '\0' is placed at the end of the string, effectively defining an end marker inside the character array.

For example

```
char instrument[64];
```

defines a string with a maximum of 63 characters (1 byte is reserved for the null). If we copy the characters a through f into the array, only the first 6 characters of the array are used, the null appears in the 7th position and the remaining elements are unused.

Arrays of Character

```
int main(void)
{
    char str[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
    printf("%s world \n", str);
    return 0;
}
```

%s in printf

memory

72
101
108
108
111
0
?
?
?

88

A string can be initialised just like any other array, but remember to put the '\0' in at the end. The length of a string is the number of characters up to, but not including, the null character.

The printf function supports strings; %s can be used to output a string. The scanf function provides the corresponding support for string input.

Language Support for Strings

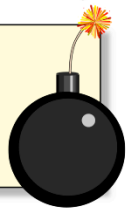
- The language supports string literals enclosed in double quotes

```
int main(void)
{
      char str[] = "world";
      printf("%s %s", "Hello", str);
      return 0;
}
```

string literals

- String literals are effectively read-only character arrays

```
void modify(char bad[])
{
     bad[0] = 'J';
}
modify("Hello");
```



89

Constant strings (or string literals) can be created using double quotation marks. This technique has already been used for the format strings in both the printf and scanf functions. A string constant can be used to initialise a string. This is effectively a character-by-character copy, from the constant into the array. This is only valid for initialization, at runtime the string must be copied character by character.

Note that a string constant can also be output from printf using %s.

C Runtime Library Functions

- **Null terminated Text Strings**
  - strlen
  - strcpy
  - strcmp
- **Binary Strings**
  - memcpy
  - memmove
  - memcmp

```
#include <string.h>

int main(void)
{
    int length;
    char s1[] = "Hi There";
    char s2[50];

    length = strlen(s1);
    printf("s1 is %d long\n", length);

    strcpy(s2, s1);

    printf("%s world\n", s2);
    return 0;
}
```

Just like s2 = s1 ;

The C runtime library has extensive support for strings. The library provides standard functions to manipulate strings, such as strlen, stcpy and strcmp.

strlen	returns the length of a string
strcpy	copies strings
strcmp	compares strings

The library not only has support for null terminated strings, but also binary strings. Binary strings use the full extent of their character array and hence do not contain null characters. Binary strings can be used to represent untyped data in memory. Some of the more common binary string routines are:

memcpy	copy memory (non overlapping regions)
memmove	copies memory (possibly overlapping regions)
memcmp	compares memory

Multidimensional Arrays

- C allows arrays of any number of dimensions to be defined:

```
type array_name[dim1][dim2]...[dimN];
```

  - e.g.

```
int table[8][6];
```

- C stores arrays in right hand dimension order
  - column first for 2D arrays
  - opposite to how Fortran handles arrays

```
[0][0]
[0][1]
[0][2]
[0][3]
[1][0]
[1][1]
[1][2]
[1][3]
```


C allows arrays of any number of dimensions to be defined. A separate set of [ ] are used for each dimension.

If you are used to programming in Fortran, note that in C multi-dimensional arrays are stored in memory with the right most dimension changing fastest. In Fortran, the left most dimension changes the fastest.

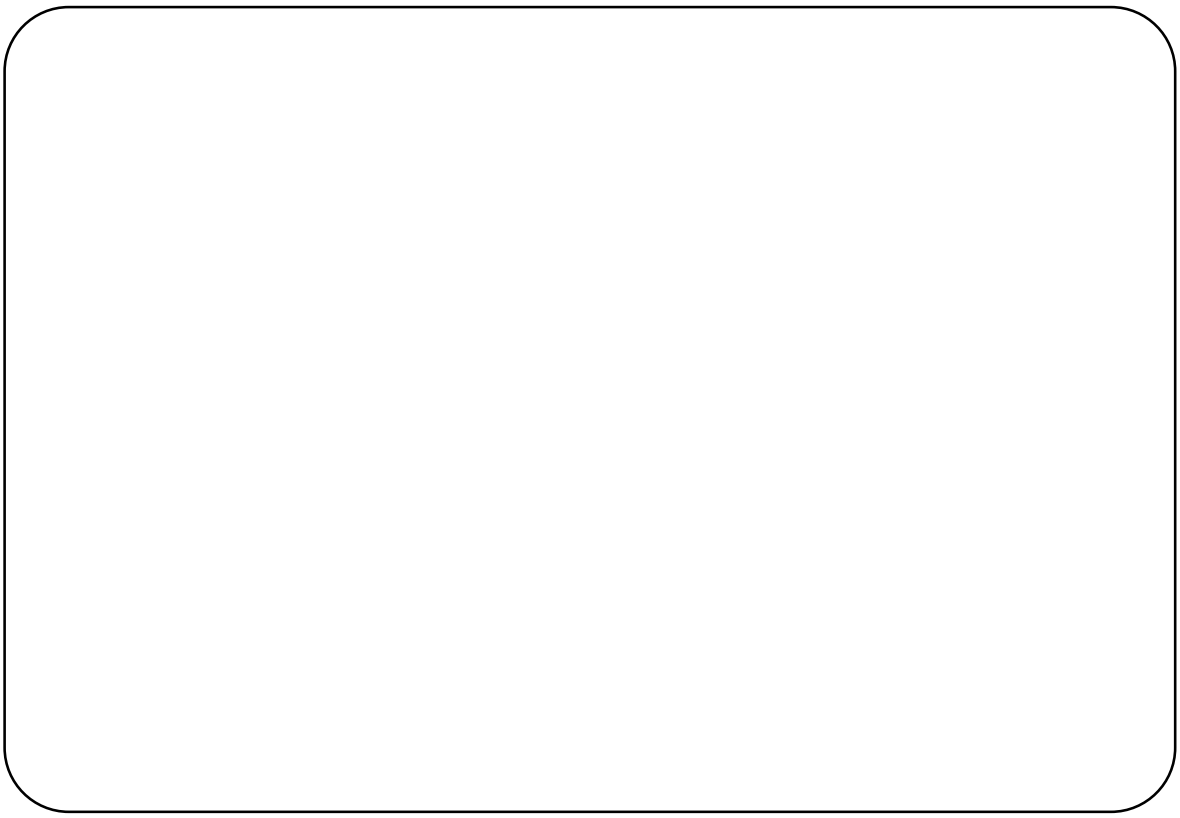
Initialising Multidimensional Arrays

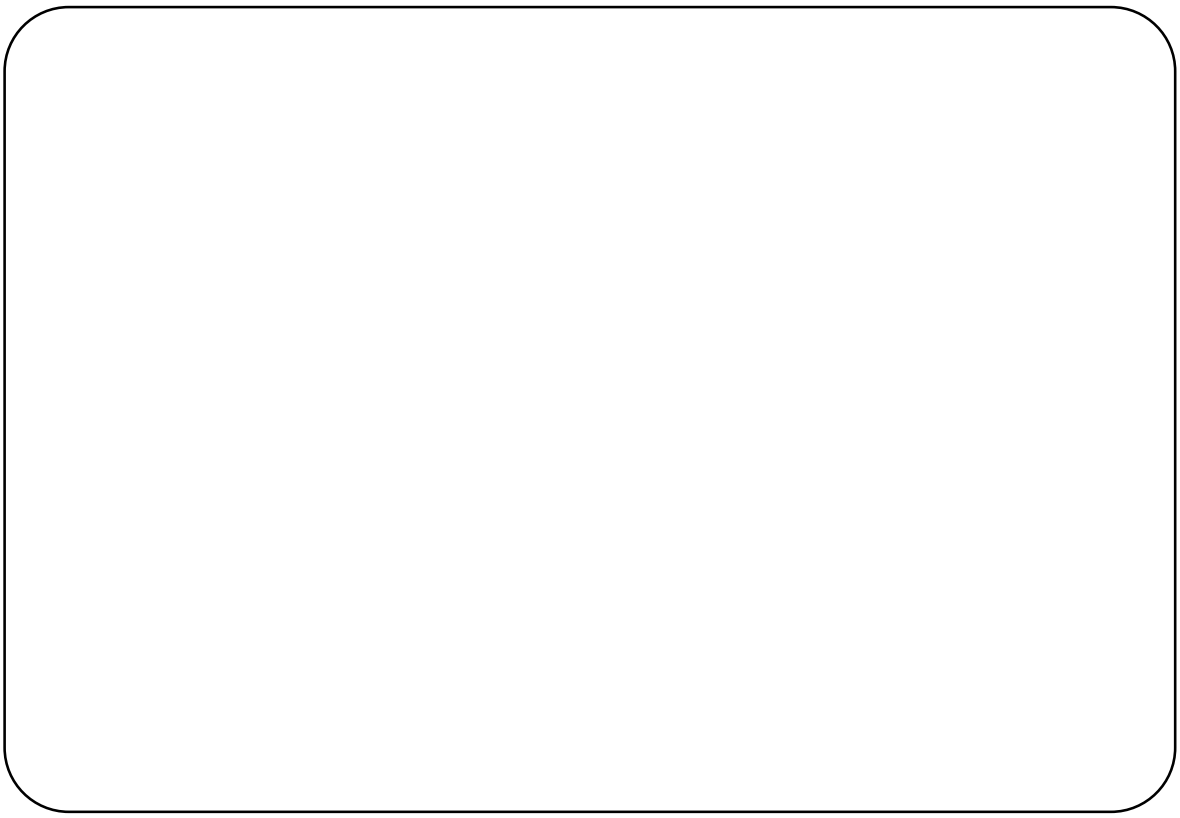
- **Multidimensional arrays can also be initialised**

```
int results[100][5] =
{
    { 75, 95, 90, 84, 80 },    /* row 0 */
    { 100, 99, 100, 98, 99 }, /* row 1 */
    ...
    { 80, 67, 85, 79, 75 }     /* row 99 */
};

double coordinates[][2] =

{
    { -2.5, 2.71 },            /* (x,y) point 0 */
    { 0.0, 0.0 },              /* (x,y) point 1 */
    { 4.0, 12.03 }             /* (x,y) point 2 */
};
```


Initialization of multidimensional arrays is performed using a set of {} for each dimension suitably nested. It is recommended that you layout initialization code as above for reasons of readability.

Note also that the compiler can calculate the row count for the second array from the initialization list.





Structures and Unions

- **Structures**
  - defining a structure
  - declaring a structure variable
  - accessing structure members
  - nested data structures
- **Unions**
  - defining a union

8

© CRS Enterprises Ltd 2001-15

Defining a Structure

- **Structure definition defines a new type**
  - **built from existing built-in or user defined types**
  - **usually defined in a header file**
- **Variables can be declared with the new type**
  - **must include the struct keyword**
  - **or use a typedef to simplify the notation**

```
struct Date
{
    int day;
    int month;
    int year;
};

struct Date myDate;
```

```
struct Card
{
    char suit;
    int index;
};

typedef struct Card CARD;
CARD myCard;
```


Structures allow you to define your own data types. Once defined, you can use your new types as you would use the built in types.

But first you must define the type using the struct keyword. Structure definitions or templates are often placed in header files, but in any case must be defined before the structure is first used.

To use the structure, you must declare a variable of the new type. Note that the declaration must include the struct keyword, for example:

struct Date myDate;

It would be simpler if we could declare variables without using the struct keyword, as in:

Date myDate;

This is allowed in C++, but not in C. The best we can do is to use a typedef to create an alias for the type. The typedef:

typedef struct Card CARD;

creates CARD as an alias for struct Card, thereby absorbing the struct keyword and then we can declare variables as in

CARD myCard;

Initialising Structs

- **Structs can be initialised at declaration time**
  - **similar to array initialization**

```
struct Date
{
    int day;
    int month;
    int year;
};

struct Date moonLanding = { 20, 7, 1969 };
struct Date manInSpace = { 12, 4, 1961 };
```


The declaration of structure variables (and constants) is like that of scalar variables. Compare the pairs of declarations below:

struct Date d ;	int d ;
struct Date days[23] ;	int days[23] ;

Initialization is similar to that used in array initialization. Both aggregates obey similar rules. Each member matches an initialiser by order and type.

As with arrays, assignment using this notation is not supported in structures. However, unlike arrays, assignment is possible. This is achieved member by member.

Accessing Structs at Run-Time

- **Structure members**
  - are accessed using the dot operat
  - holiday.year
- **Complete structure assignm**
  - christmas = holiday

```
struct Date
{
    int day;
    int month;
    int year;
};

int main(void)
{
    struct Date holiday;
    holiday.day = 25;
    holiday.month = 12;
    holiday.year = 2007;

    struct Date christmas;
    christmas = holiday;
}
```


Although members are declared sequentially, there is no guarantee that the compiler keeps to this order internally. Access is achieved using the dot or 'structure member' operator.

Complete structure operations are supported, for example

christmas = holiday

Input/Output

- Input using pointers
- Output using members
  - do not use pointers

```
struct Date
{
   int day;
   int month;
   int year;
};
```

```
int main(void)
{
    struct Date myDate;
    printf("Enter date\n");
    scanf("%i %i %i", &myDate.day,
                      &myDate.month,
                      &myDate.year);
    printf("Date is %i/%i/%i\n", myDate.day,
                                 myDate.month,
                                 myDate.year);
    return 0;
}
```


Input and output for structures is similar to built in data types, except the structures must be input or output a component at a time. It is not possible to work with complete structures.

Combining Structures

- **Structures can be nested**

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

struct Rectangle {
    struct Point topLeft;
    struct Point bottomRight;
};
...
```

```
...
int main(void)
{
    struct Rectangle r;
    r.topLeft.x = 0;
    r.topLeft.y = 0;
    r.bottomRight.x = 1000;
    r.bottomRight.y = 1000;

    return 0;
}
```


Structures can be nested. The ability to nest aggregates means that the programmer can obtain quite close relationships between the program data and the 'real-world' data, both in name and data type.

In the example above, a Rectangle structure is defined in terms of two Point structures. Note that to access the components of the rectangle:

r.topLeft.x = 0;

r.topLeft.y = 0;

r.bottomRight.x = 1000;

r.bottomRight.y = 1000;

two dot operators are used, reflecting the two levels of nesting for the Rectangle structure.

Arrays of Structures

- **We can represent a hand of 5 cards**

```
struct Card
{
    char suit;
    int index;
};

struct Card hand[5];

hand[0].suit = 's';  hand[0].index = 13;
hand[1].suit = 'h';  hand[1].index = 12;
hand[2].suit = 's';  hand[2].index = 4;
hand[3].suit = 'd';  hand[3].index = 7;
hand[4].suit = 's';  hand[4].index = 1;
```


An array of structures is defined in a similar way to built in data types. Remember to include the struct keyword unless you define an alias using a typedef.

Code to access individual components of the structure is shown above.

Initialization

- **We can initialise an array of structures**
  - **two sets of {}**

```
struct Card
{
    char suit;
    int index;
};

struct Card hand[5] = {
                       {'s',13},
                       {'h',12},
                       {'s', 4},
                       {'d', 7},
                       {'s', 1}
                      };
```


Arrays of structures can be initialised. The syntax is similar to the initialization of two dimensional arrays - two nested sets of {} are required.

Passing Structures to Functions

- **Structures**
  - **can be passed as function argum**

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

typedef struct Point Point;
```

```
Point Invert(Point p)
{
    int temp = p.x;
    p.x = p.y;
    p.y = temp;
    return p;
}

int main(void)
{
    Point point1 = { 100, 200 };
    Point point2 = Invert(point1);
    return 0;
}
```


When structures are passed to functions, the normal call by value semantics are used. This means the function receives a copy of the structure.

It is often desirable to pass the address of a structure to a function instead of the structure itself, for efficiency reasons. Structures can be very large, and call by value could be expensive in terms of time and space. However, a pointer to a structure will only consume 4 bytes (on a 32 bit machine). Remember that if you do pass a the pointer to a structure, you are simulating call by reference and the function can modify the original structure. If this is undesirable, and you don't want to pass a copy of a large structure, then try passing a const pointer instead:

Point Invert(const Point* p) { ... }

This is not only efficient, but the compiler will prevent modifications to p.

unions

- A union is a variable that may hold data of different types and sizes at different times

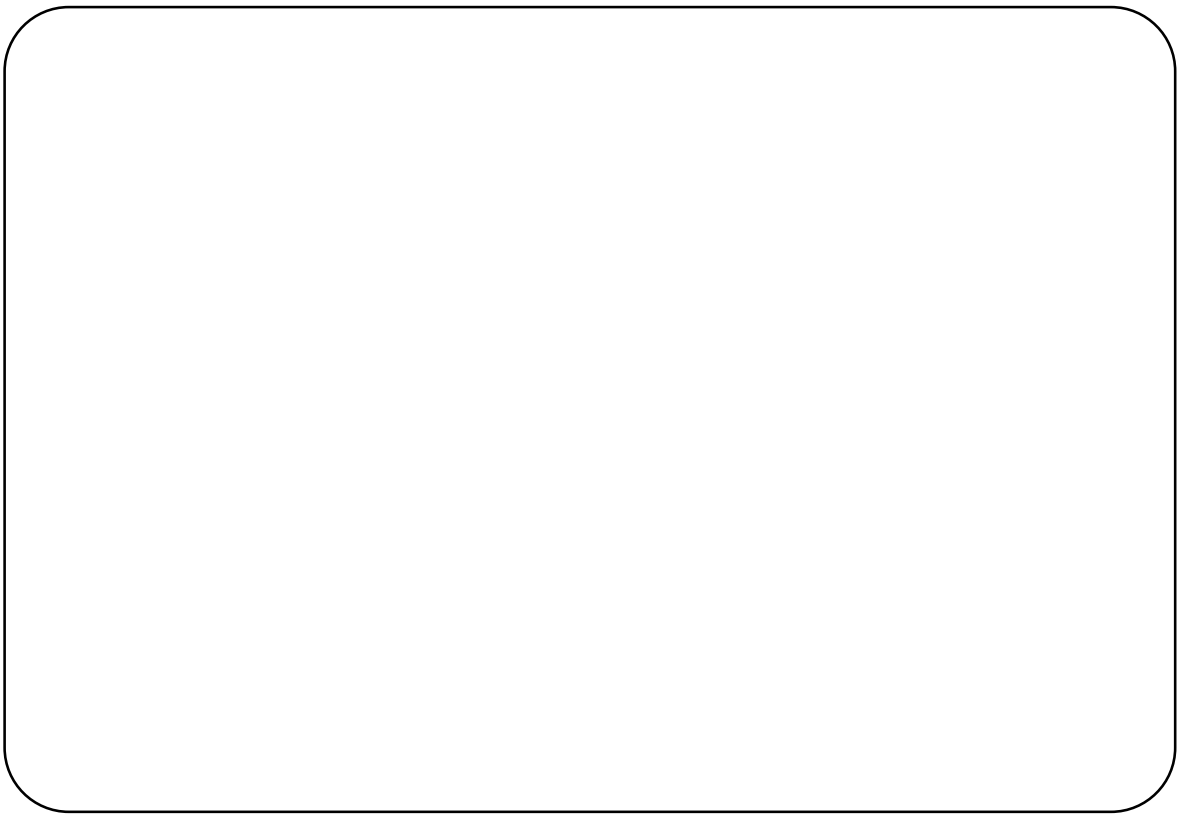
```
#include <stdio.h>

union Variant {
    char initial;
    char name[10];
    int  worksNumber;
};

int main(void)
{
    union Variant theWorker;
    theWorker.initial = 'S';
    strcpy(theWorker.name, "Smith");
    theWorker.worksNumber = 31572;
}
```

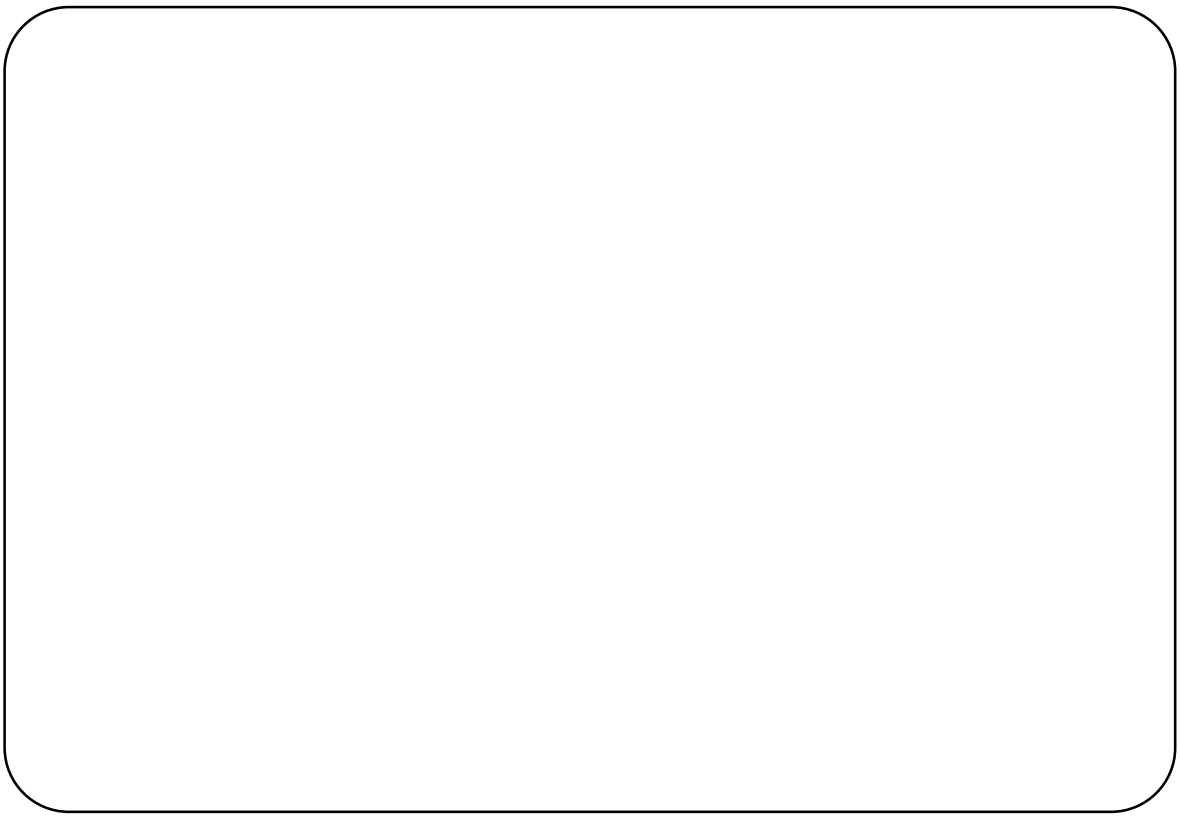
- Unions syntax
  - same as structures
- All members share storage
  - only one type is valid at any instant

A union is a variable that may hold data of different types and sizes at different times. It follows the syntax of a struct, but has different semantics.

The distinct feature of a union is that all the components share the same storage. This implies that only one component can be used at any one time. The idea of a union is to allow a variable to behave as a different type at different times. This is tantamount to breaking the strong typing of C. From a design prospective, unions are of dubious value.

Unions are highly efficient in terms of storage and were very prevalent in the 70s and 80s when memory was in short supply. Nowadays, unions are an anachronism; the memory saved in miniscule and the potential for generating obscure code is all too real.





Pointers, Arrays and Functions

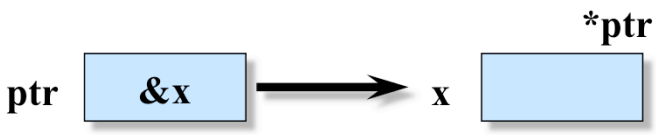
- **Arrays**
  - relationship to pointers
  - pointer notation
- **Manipulating Pointers**
  - pointers to arrays
  - indexing
  - pointer arithmetic
- **Functions**
  - call by value
  - call by reference

9

© CRS Enterprises Ltd 2001-15

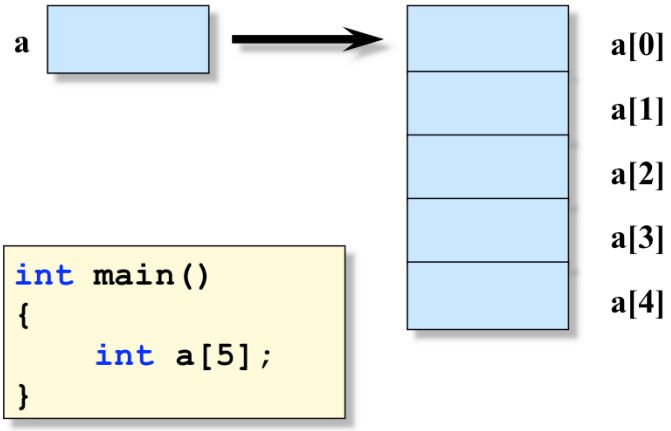
Pointers - Review

```
int main()
{
    int x = 100;
    int* ptr;

    ptr = &x;
}
```



In this chapter we will examine the close relationship between pointers and arrays in C. This relationship allows pointers to be associated with arrays using [].

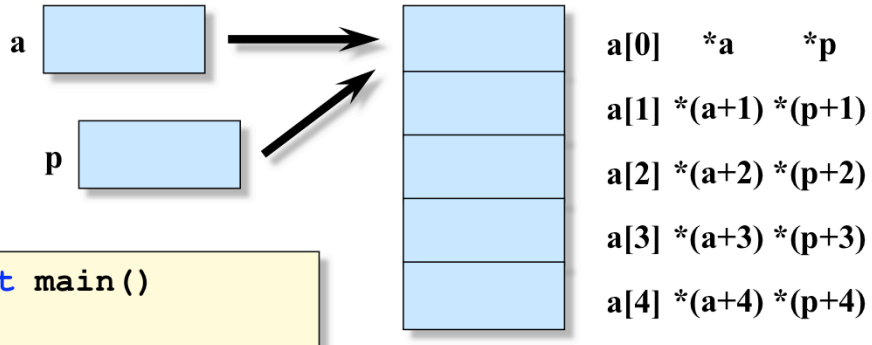
Pointers and Arrays




```
int main()
{
    int a[5];
}
```


There are two major reasons why arrays are closely linked with pointers. Firstly, the array name is a constant pointer that contains the address of its first element. Secondly, array elements are placed in contiguous memory.

Pointer Notation




```
int main()
{
    int a[5];
    int* p;

    p = &a[0];
}
```

 110

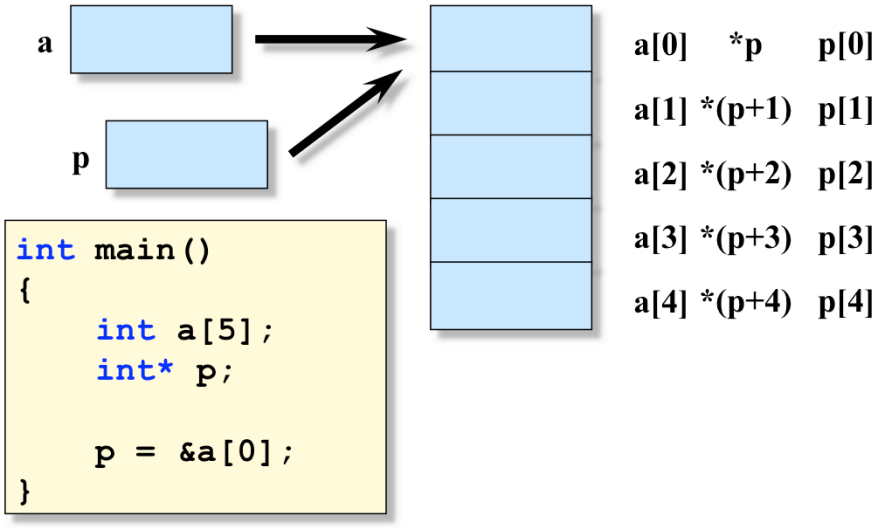
As stated previously, an array's name is a const pointer (an r-value). This implies that in addition to the standard array notation for accessing elements of the array, this pointer can access array elements using *a, *(a+1), *(a+2) etc. If you use a pointer to point at the array then the pointer can "see" the entire array using *p, *(p+1), *(p+2) etc.

Note that pointer arithmetic takes into account the size of the data being dereferenced. Thus for the integer pointer p,

*(p+3)

means the contents of the address at p + 3 * sizeof(int) or p + 12 bytes.

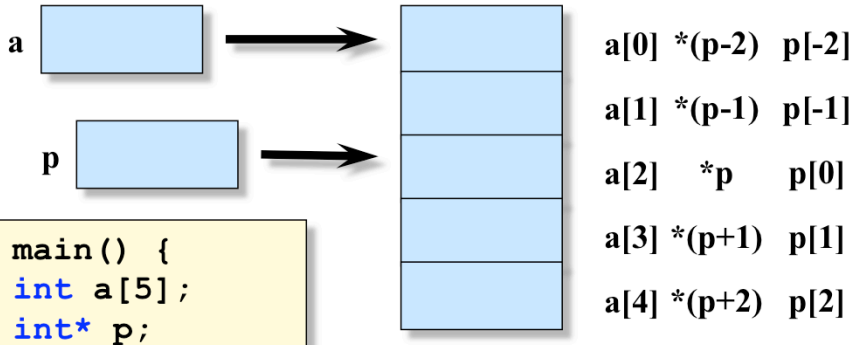
Array Notation




```
int main()
{
    int a[5];
    int* p;

    p = &a[0];
}
```


By analogy with arrays (const pointers), a true pointer can also access an array using [] notation as in p[0], p[1], p[2] etc.

Moving Arrays?




```
int main() {
    int a[5];
    int* p;

    p = &a[0];
    p++;
    p++;
}
```


Recall that the name of an array is a const pointer (r-value). However if we use a true pointer to align with the const pointer, this new pointer is not const (l-value). This means that the new pointer can be incremented or decremented. By incrementing the pointer twice as in the example above, the array appears to move in memory and p[0] is now an alias for a[2].

The implications of all this are that you cannot distinguish between pointers and arrays at runtime, apart from the array pointer cannot appear on the left side of an assignment. This in turn implies that negative indices may well be valid array indices at run time.

Of course at compile time arrays allocate storage for data at the end of the array pointer, whereas pointers do not.

Indexing

```
int a[6] = { 2, 3, 5, 7, 11, 13 };
int* p  = a;
```

*p++	(*p)++
Equivalent to *(p++)	( ) dictates precedence
Yields *p increments p	Yields *p increments *p

When accessing arrays you can use pointer notation (* operator) or array notation ([] notation). The expression

*p++

is often used inside a loop to step through an array extracting each array element in turn. Operator precedence dictates that this expression yields the value *p and p is incremented as a side effect. Contrast this with

(*p)++

This expression yields *p and increments *p - not much use for stepping through an array.

Pointer Arithmetic

```
#include <stdio.h>

int main(void)
{
    int a[6] = { 2, 3, 5, 7, 11, 13 };
    int* p = a;
    int* end = a + 6;
    int sum = 0;

    while (p < end)
        sum += *p++;
    return 0;
}
```


This examples uses the pointers to step through an array. The key line is the statement

sum += *p++;

This statement does the following:

the RHS yields *p and hence the statement's effect is sum += *p

the side effect operator increments the pointer after the += operator has completed

By the end of the loop, sum will contain the sum of the integers stored in the array. The same effect can be achieved without pointers:

```
int i;
for(i = 0; i < 6; i++)
{
   sum += a[i];
}
```

Although many books imply that using pointers is superior to using arrays, there is no guarantee which method will produce the faster code. On some machines using

Call By Value

- **Call by value**
  - **means the copies get swapped and not the originals**
  - **must use pointers to access original data inside the function**
  - **in C++ you can also pass by reference, but this is not available in C**

swap.c

```
void swap(int, int);

int main(void)
{
    int x = 100;
    int y = 200;

    swap(x, y);
    return 0;
}

void swap(int a, int b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
}
```


All parameters passed to functions are passed as copies of the original data. This is called pass by value. The idea is to prohibit the function modifying the original data. However sometimes, as in a swap function, we do want to modify the original data. If we try with call by value the swap fails and only the copies get swapped.

To access the original data, pointers must be used (see subsequent slides). This is called call by reference.

Simulated Call By Reference

- **Pass pointers to original data**
  - **modify the prototype and**
  - **modify the call**

```
void swap(int*, int*);

int main(void)
{
  int x = 100, y = 200;

  swap( &x , &y );
}
```

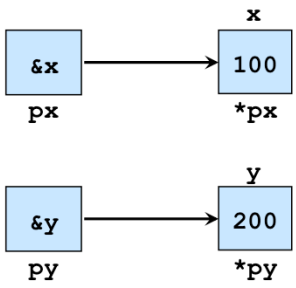
prototype

pointers

By passing pointers to the original data we can get swap to modify the original data. In the example above, we call swap with arguments &x and &y; the addresses of x and y. The prototype indicates that the swap function will receive the addresses of two integers. Of course we still need to write the swap function!

Modifying the Function

- **The function makes changes to the original data**
  - **using the pointers**



swap.c

```
void swap(int*, int*);

int main(void)
{
   int x = 100, y = 200;
   swap(&x, &y);
   return 0;
}

void swap(int* px, int* py)
{
    int temp;
    temp  = *px ;
    *px   = *py ;
    *py   = temp ;
}
```


The definition of the swap function completes the picture. The two parameters px and py are declared as 'pointers to int'; they are automatically initialised with the values passed by the call: &x and &y.

The function body makes use of the * operator to get hold of the data pointed to by px and py. The expressions *px and *py are therefore aliases to x and y respectively. We would like to write

temp = x;

x = y;

y = temp;

but x and y are not in scope in the function. However their aliases *px and *py are in scope, so we actually write

temp = *px;

*px = *py;

*py = temp;

Returning Pointers

```
int* getMax(int array[], int size);

int main(void)
{
    int numbers[5] = { 3, 20, 5, 22, 7 };
    int* pMax;
    pMax = getMax(numbers, 5);
    *pMax = 0;    // Set the max to zero
    return 0;
}
```

```
int* getMax(int array[], int size)
{
    int* pMax;
    int i;

    pMax = &array[0];
    for (i = 1; i < size; i++)
        if (*pMax < a[i]) pMax = &a[i];
    return pMax;
}
```


Returning a pointer is like returning an int, char, etc. The return type must be properly specified in the prototype and definition, i.e. int *.

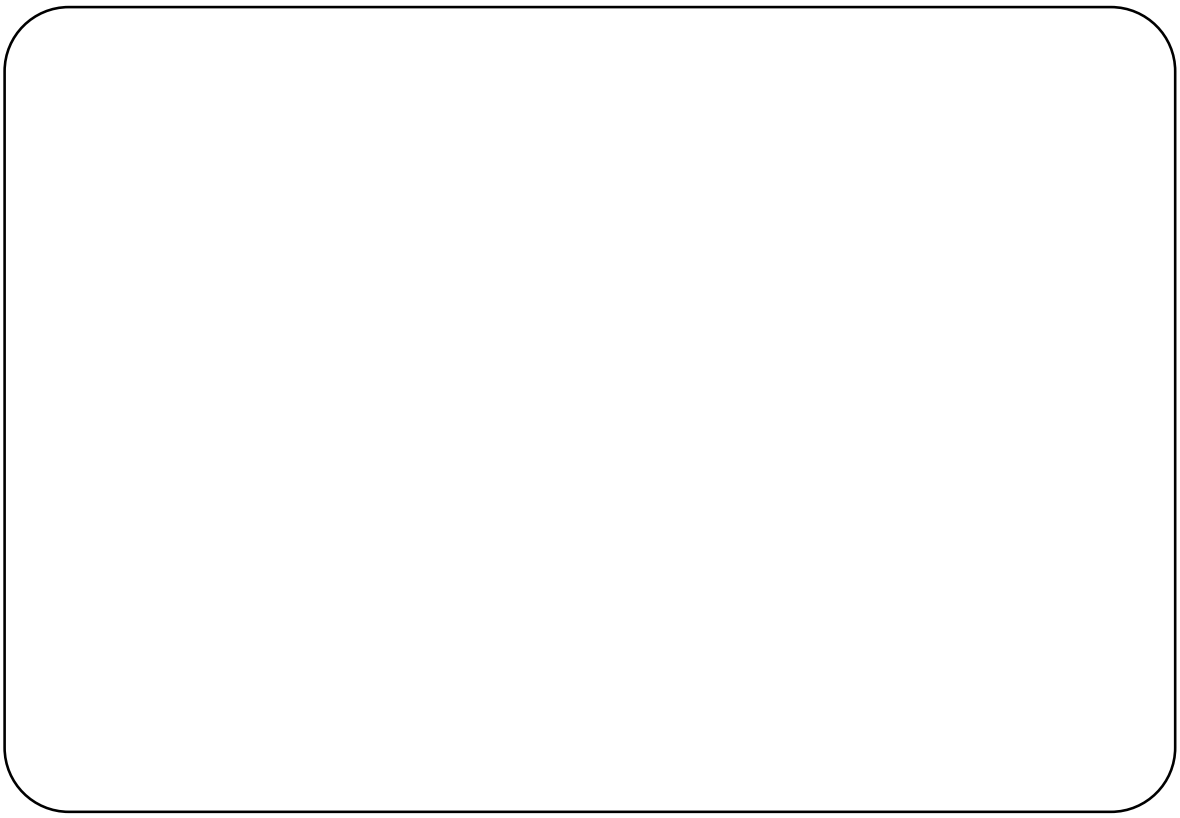
The only problem in returning pointer types is that the value of the returned address has to exist. The address must be accessible to the calling function's environment and not, for example, the address of a local variable which will cease to exist on return from the function. In our example shown above, the address is part of the local array, marks. This is perfectly acceptable because the array will continue to exist after the call completes.

Pointers, Arrays and Functions

```
#include <stdio.h>

int total(int*, int);

int main(void) {
    int a[ ] = { 2, 3, 5, 7, 11, 13 };
    int sum = total(a, 6);
    return 0;
}

int total(int* pStart, int size) {
    int sum = 0;
    int* pEnd = pStart + size;

    while(pStart < pEnd)
        sum += *pStart++;
    return sum;
}
```


The example shows how one of earlier examples can be recoded using a function. This combines the use of pointers, arrays and functions into a single example.

Note that when passing an array to a function, you are really passing a pointer (l-value) to its first element. Since there is no way of the function deducing the size of the array, this must be passed as a separate parameter.



Pointers and Structures

- **Structures**
  - declaring structure pointers
  - the structure-pointer operator
- **Passing structures to functions**
  - pass by value
  - pass by reference
- **Dynamic data structures**
  - linked lists
  - trees

10

© CRS Enterprises Ltd 2001-15

Pointers to Structures

- **Structures can be passed as function parameters**
  - **pass by copy**
  - **large structure => inefficient**
- **Pass a pointer to the structure**
  - **more efficient than passing by value**
  - **pass by reference**
- **Pointers to structures**
  - **used to access structures on the heap**
    - **essential for run time structure allocation**
  - **used to create advanced data structures**
    - **such as linked lists**
    - **binary trees**

Structures can be passed as function parameters. The default mechanism is pass by copy (call by value), but for a large structures this can be inefficient. This leads us to the idea of passing a pointer to the structure (call by reference).

In practice, we employ this technique frequently. For example, when we consider dynamic memory allocation on the heap (see later), we will see that we are forced to use pointers to access dynamically allocated structures. Furthermore, pointers to structures are used when we design complex data structures such as linked lists and binary trees.

Structures - A Review

- The '.' operator provides member access capability

```
struct Date
{
    int day;
    int month;
    int year;
};

int main(void)
{
    struct Date today;

    /* set today's date */
    today.day = 17;
    today.month = 4;
    today.year = 2008;
    return 0;
}
```

First, review the way we use structures.

Firstly we define the structure - a template or blueprint from which structure variables are created.

Next we declare and optionally initialise our variables - each variable possesses the same set of components.

Finally we use the dot operator to access the data members.

Declaring Pointers to Structures

```
struct Date
{
    int day;
    int month;
    int year;
};

int main(void)
{
    struct Date today;
    struct Date* pDate = &today;

    (*pDate).day = 17;
    (*pDate).month = 4;
    (*pDate).year = 2008;
    return 0;
}
```

pointer to a structure

When working with pointers to structures we need to employ the * operator for dereferencing. This leads to code such as

(*pDate).day = 17;

(*pDate).month = 4;

(*pDate).year = 2008;

regarded by some as rather clumsy. The brackets are required because of precedence.

As we will discover overleaf, C provides an alternative mechanism for writing the above code.

The Structure-Pointer Operator

- **Pointers to structures**
  - **used so often in C that a special operator exists**
- **The -> operator allows expressions like**

```
(*struct_ptr) . member_name
```

  - **to be written like**
    -

```
struct_ptr -> member_name
```


The arrow operator can be used to simplify the previous example. The operator absorbs the *, the () and the dot operator. Many programmers prefer using this operator than its more verbose alternative.

It is important to realise that as far as the compiler is concerned, both notations are equally valid. The former notation using the *, () and dot operator is more common in object oriented programming and you will see it widely used in C++. The choice is yours.

Note that the arrow operator is sometimes called the structure-pointer operator, the pointer-to-structure access operator or the crow's-foot operator.

Using the -> Operator

- **access structure components**
  - **using the -> operator**

```
struct Date
{
    int day;
    int month;
    int year;
};

int main(void)
{
    struct Date today;
    struct Date* pDate = &today;

    pDate->day = 17;
    pDate->month = 4;
    pDate->year = 2008;
}
```

```
(*pDate).day = 17;
(*pDate).month = 4;
(*pDate).year = 2008;
```


The example shown above illustrates the syntax of the -> operator. Compare this with the * () and dot alternative.

Passing Structures to Functions

```
struct Date
{
    int day;
    int month;
    int year;
};

void print(struct Date*);

int main(void)
{
    struct Date today = { 17, 4, 2008 };
    print(&today);
}

void print(struct Date* pDate)
{
    printf("Day = %i\n", pDate->day);
    printf("Month = %i\n", pDate->month);
    printf("Year = %i\n", pDdate->year);
}
```

pass a pointer to the structure

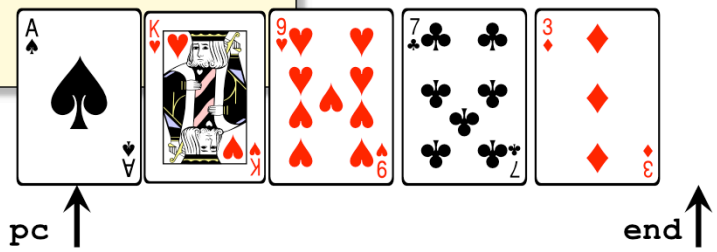
This is a more realistic example. Note the argument type in both the prototype and the definition. This is very similar to the syntax used in our swap call-by-reference example.

The function has complete freedom to access the today structure via its address since the structure is effectively passed by reference. Since this is a print routine, we are only accessing the structure to read its details for display purposes, but we could have modified the structure if desired.

Pointers to Arrays of Structures

```
...
int count;
struct Card* pc = &hand[0];
struct Card* end = pc + 5;

for (count = 0; pc != end; ++pc)
{
    if (pc->suit == 'h')
    {
        count++;
    }
}
...
```

How can we write this using a function?



The arrow operator can be used with arrays of structures. The technique is to move the pointer through the array with

++pc

and continue while the pointer has not gone beyond the end of the array

pc != end

Note the way the pointer end is calculated

end = pc + 5

relying on pointer arithmetic to take into account the size of each card.

Passing Arrays of Structures

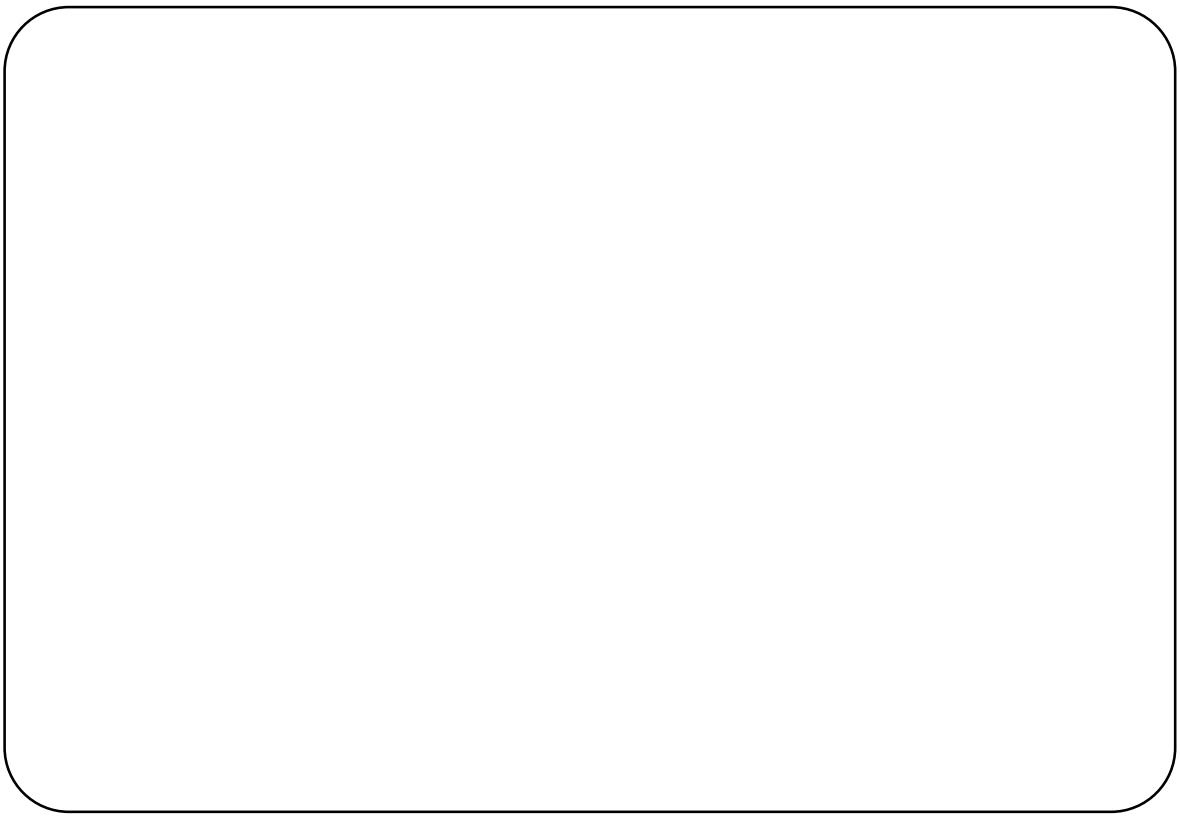
```
typedef struct Card CARD;

int main(void)
{
    CARD hand[SIZE];
    int count;
    count = countHearts(&hand[0], 5);
}
```

```
int countHearts(CARD array[], int size)
{
    int count = 0;
    int i;

    for (i = 0; i < size; ++i)
    {
        if (array[i].suit == 'h')
        {
            count++;
        }
    }
    return count;
}
```

```
int countHearts(CARD* cp, int size)
{
    int count;
    CARD* end = cp + size;

    for (count = 0; cp != end; ++cp)
    {
        if (cp->suit == 'h')
        {
            count++;
        }
    }
    return count;
}
```


Finally, contrast the two techniques of using arrays (left hand example) or using pointers (right hand example) to implement the countHearts function. Both techniques are equally valid.

You should be aware that even in the presence of an optimizing compiler, the two solutions will probably produce different code. Which method produces the most efficient code will be dependent on the compiler used. However, it is extremely unlikely that either technique will differ much in performance, so the choice is one of programming style. Choose the method you fell is easier to read, understand and maintain. You can be sure that someone else will disagree.



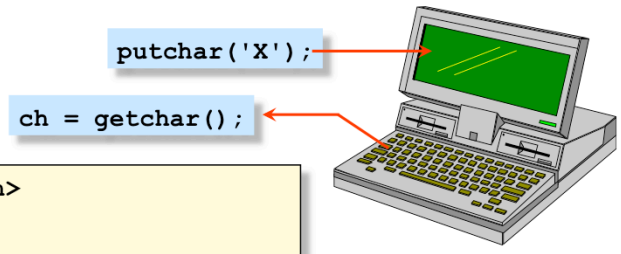
Input and Output

- **Console I/O**
  - getchar, putchar
- **Command Line**
  - argc, argv
- **File I/O**
  - fopen, fclose
  - fputc
  - fprintf, fscanf
  - fseek

11

Stdin and Stdout

- **Use getchar()**
  - to read from stdin
- **Use putchar()**
  - to write to stdout




```
#include <stdio.h>
void echo(void)
{
    int ch;

    while ((ch = getchar()) != EOF)
        putchar(ch);
}
```


Two of the simplest I/O functions are getchar and putchar. These functions read and write to STDIN and STDOUT respectively.

Perhaps surprisingly, the return from getchar and the argument to putchar are both ints. If these functions worked with 8 bit chars there would be no safe way to return an end of file (EOF) indicator. Under normal operation these functions work with ints in the range 0-255, but use -1 to represent EOF. As you can see from the example you can the symbol EOF directly; this is because EOF is #defined in <stdio.h>.

Note: actually both getchar and putchar are not functions at all; they are in fact preprocessor macros made to look like functions. In <stdio.h> they are #defined in terms of getc and putc.

Command-Line Arguments

- **Pass arguments to a program when it begins to execute**
- **argc**
  - **number of arguments**
- **argv[ ]**
  - **array of pointers to argument strings**

alternative syntax

```
int main(int argc, char** argv)
{ ...
```

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int i;
    for (i = 0; i < argc; i++)
    {
        printf("%s\n", argv[i];
    }
}
```


The C programming language was written originally to code the Unix operating system. Nearly, all the Unix commands can be run from the command line, possibly with arguments. Hence C requires a mechanism to pass command line arguments to your program. This is achieved as shown above.

The argv pointer accesses the command-line arguments as strings. It defines an array of char pointers, one for each input argument (string). The number of arguments in the array is defined in argc.

A third argument, usually called envp, is available. It has the same structure as argv, but contains strings that hold the values of the environment variables.

Writing to a File

- **Open a file using fopen**
  - **"w+" means open for read and write and truncate contents**
  - **"r+" means open for read and write, but do not truncate contents**
  - **"a" for append**

```
#include <stdio.h>
int main(void)
{
    int i;
    char ch = 'A';

    FILE* fptr = fopen("myfile.txt","w+");
    for(i = 1; i <= 26; i++) {
        fputc(ch, fptr);
        ch = ch + 1;
    }

    fclose(fptr);
}
```


All input and output, whether using console, files, communication ports, etc. is performed by library routines. The standard library, defined by <stdio.h> contains numerous functions to perform I/O.

Using the library, you must first open a file before you can read or write to it. open returns a file handle (FILE*) which is used in all subsequent file I/O. You should close a file when you are finished with it. Failure to do so will tie up resources in memory until you exit your program; the operating system will close any files you failed to close when your program exits.

The open function uses several modes of accessing a file. These modes can be quite confusing. If you use and "r" or a "w" the mode is read or write, but if you use a "r+" it means read plus write and similarly "w+" means write plus read. In fact "w" also implies that the file will be truncated on opening (i.e. its contents deleted) and this is the essential difference between "r+" and "w+"; both open a file for reading and writing, but the latter also truncates the file. Other modes include "a" for appending, "b" for binary mode and "t" for text mode. Binary and text mode are only used for Windows; Unix does not distinguish between binary and text files.

Note: both Unix and Windows have other routines defined, outside the scope of the standard library, that allow memory mapped I/O (e.g. mmap in Unix; CreateFileMapping in Windows). With these routines, files do not have to be opened before they can be used.

Formatted I/O

- fprintf
  - uses the same format strings as printf

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

int main(void)
{
   FILE* fptr;
   int x = 100;
   double y = 3.14159;
   char z[ ] = "the quick brown fox ...";

   fptr = fopen("myfile.txt","w+");
   fprintf(fptr, "%i %lf %s \n", x, y, z);
   fclose(fptr);
}
```


Several functions are available for formatted output, but fprintf is the most widely used. As its name implies it behaves exactly the same as printf. It has one additional parameter - the file pointer returned by fopen.

Similarly the function fscanf can be used for formatted input.

Block I/O

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  Money m[3];
  FILE* fptr;

  m[0].euros = 50;    m[0].cents = 6;
  m[1].euros = 25;    m[1].cents = 9;
  m[2].euros = 0;     m[2].cents = 12;

  fptr = fopen("myfile.dat","w+b");
  fwrite(&m[0], sizeof(Money), 3, fptr);
  fclose(fptr);
}
```

```
typedef struct {
   int euros;
   int cents;
} Money;
```


With file I/O it is convenient to have routines that read and write large blocks of data to and from files. Typically, you will define a large buffer in memory to represent and array of structures. Using fwrite you can write this buffer directly to a file.

fwrite takes 4 parameters:

1st parameter:	pointer to the buffer
2nd parameter:	the size of each struct to be written to the file
3rd parameter:	the number of structs to be written to the file
4th parameter:	the file pointer

The actual number of bytes written to a file will be the product of the second and third parameters.

Random Access

- use fseek
  - to position file pointer

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

int main(void) {
   FILE* fptr;
   int i;
   char theString[8 + 1];
   fptr = fopen("myfile.dat","w+b");
   for(i = 0; i < 10000; i++) {
      fprintf(fptr, "%8i", i);
   }
   fseek(fptr, 2500 * sizeof(int), SEEK_SET);
   fscanf(fptr, "%s", theString);
   fclose(fptr);
}
```

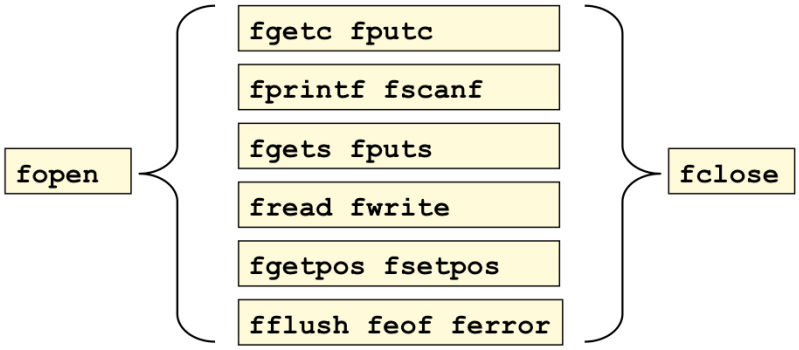
 137

All file I/O considered so far works by sequentially accessing a file. In fact the library maintains an internal pointer that keeps track of where you are in a file: the file position indicator. Random access file I/O works by giving you access to the file position indicator. You can change its value using fseek without performing any I/O. For large files you can use fseek to jump around in a file and read just the parts you are interested in, without having to read all the intervening data.

Remember that accessing disk is a very slow operation compared with memory access. When you use fseek you are simply changing the in memory file position indicator and hence fseek is very fast operation.

Buffered File Access

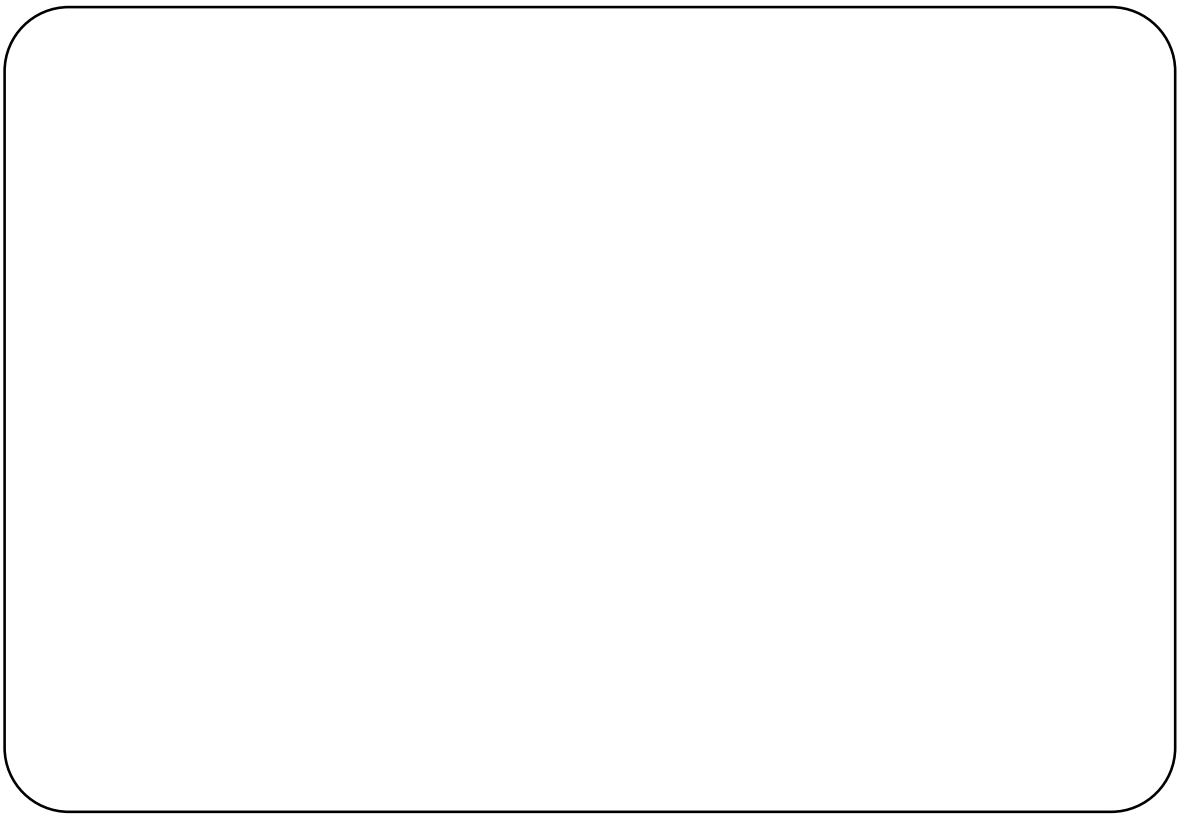
- **C Standard Library**
  - **many functions for I/O**
  - **all use buffered I/O**
  - **must open file with fopen and close with fclose**

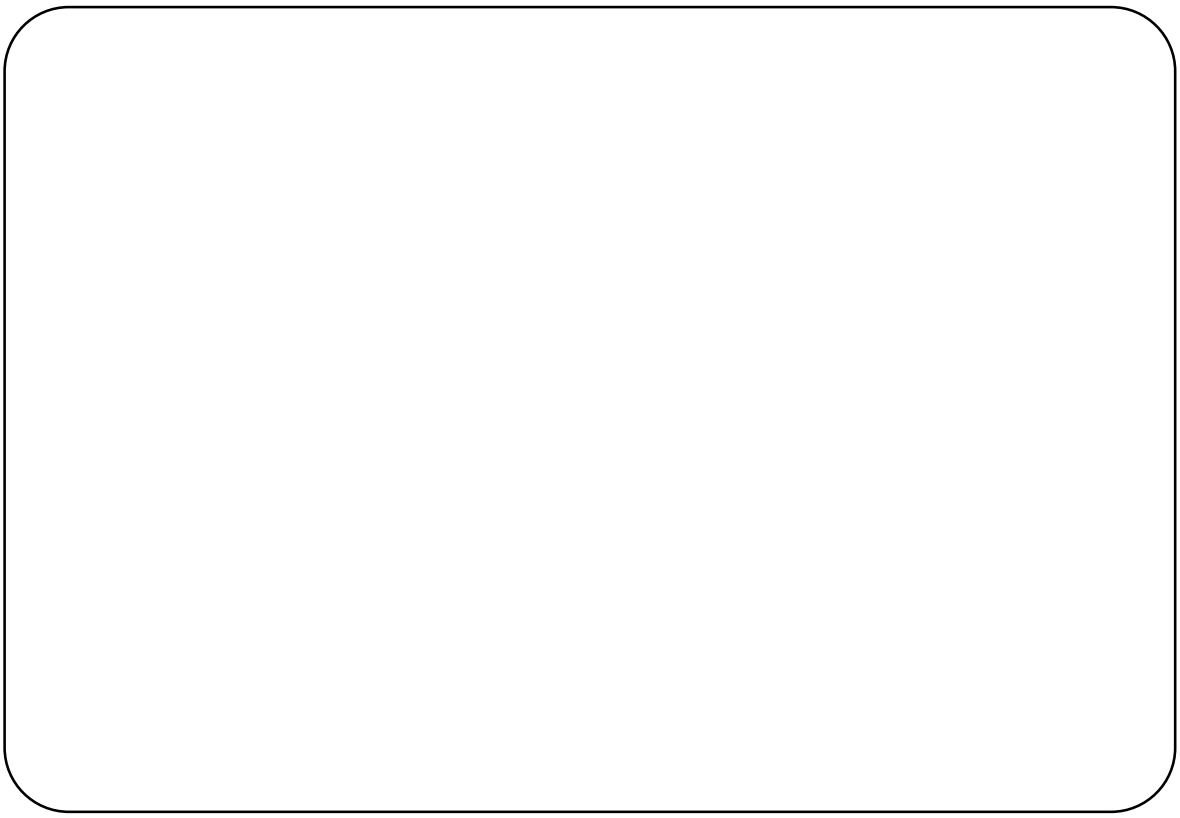


 138

Above is a synopsis of the C standard library. All these routines take a file handle as an argument. A more complete list is given below:

fopen, fclose	must be used
freopen	as fopen, but with user-supplied file handle
fflush	forces data to be read/written to the input/output stream
fscanf, fprintf	file versions of scanf and printf
fgetc, fputc	file versions of getchar and putchar
fgets and fputs	file versions of gets and puts (work with strings)
ungetc	pushes back a character obtained by getc into I/O buffer
fread and fwrite	file I/O of arrays of structures
fseek and ftell	random access file I/O
clearerr, feof, ferror	error handling using <error.h>





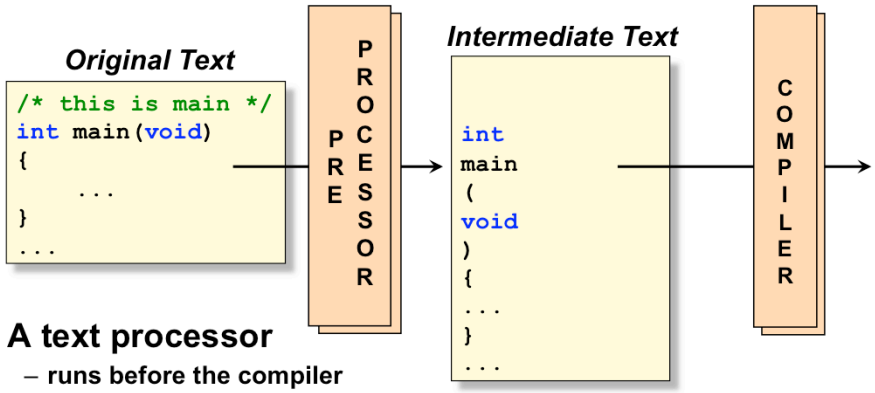
The Preprocessor

- **Basic Features**
  - Constants
  - Macro substitution
  - File inclusion
  - Conditional compilation
- **Advanced Features**
  - string sizing operator
  - #pragma

12

© CRS Enterprises Ltd 2001-15

What is a Preprocessor?



- **A text processor**
  - runs before the compiler
  - simple textual replacement
  - a tokeniser
  - strips out comments, ...

The preprocessor is separate from the compiler and performs limited text processing. You can think of the pre-processor as the first parse of the source code. Its capabilities are discussed in this chapter.

The main purpose of the pre-processor is to translate text according to a set of rules. The output from the preprocessor, becomes the input to the compiler. Normally the pre-processor output is held temporarily in memory and then discarded at the end of the compilation phase. However, using various compiler options, you can save the output as a temporary file with a .i extension or display the output on STDOUT.

Preprocessor Directives

- **Preprocessor directives may be embedded in C code**
  - **All directives start with #**
- **Symbolic Constants**
  - **#define**
- **Include files**
  - **#include**
- **Conditional Compilation**
  - **#ifdef, #ifndef, #else, #elif, #endif**
- **String Concatenation**
  - **##**
- **Quoting**
  - **#**

```
#include ...
#define ...
#ifdef ...

int main(void)
{
    ...
}
```


Pre-processor commands can be placed anywhere in the source file. Without exception, they begin with the # character. Syntax is similar, but not identical, to C. The pre-processor is regarded as part of the C environment and is controlled by the ISO standard. The following directives are supported:

#define and #undef

#include

#if #elif #else #endif

#ifdef #elif #else #endif

#ifndef #elif #else #endif

#line

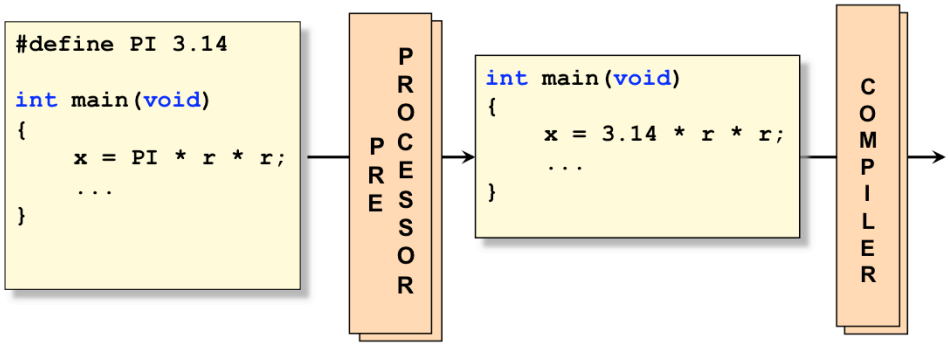
#pragma

#error

As well as these, there are other facilities: trigraphs, predefined identifiers and the two operators, # and ##.

#define

- A #define control line can be used to give a symbolic name to some text



- By convention, symbolic constants are in upper case

One of the main uses of the pre-processor is to give symbolic name to constants. By convention, preprocessor symbols are written as upper case. In the example above, the symbol PI is replace by the text 3.14. Note that the pre-processor only works with text; it is unaware of the concept of numbers, arrays and most other C artifacts.

The #undef directive is used to wipe out a previously #defined name. For example:

```
#undef PI
```

This is usually used when the symbol is to be given a new value in the source that follows. The symbol can be any legal C identifier.

Note: preprocessor directives are terminated by the end of line unless the \ continuation character is used.

More #defines

```
#define BUFFER 100
int main(void)
{
  double buffer[BUFFER];
  int i;
  ...
  for (i = 0; i < BUFFER; i++)
    buffer[i] = 0.0;
  ...
}
```

```
#define   VAT 17.5
#define   TAX 1.5 + VAT
int main(void)
{
  double amount;
  ...
  amount = TAX * 21.50;
  ...
}
```

what does this expand to →

By using the pre-processor to define symbolic constants, it becomes much easier to maintain code. For example suppose a buffer size is used repeatedly throughout a program. If the buffer size needs to be changed, then only the one #define needs to be modified.

Because the pre-processor effectively performs a single parse, the scope of its #define tokens are global and take effect on the line after the definition. This means that you may define symbols using symbols already defined. However, you must be careful with precedence. Brackets are nearly always required. For example, with the definition:

```
#define  TAX  1.5 + VAT
```

Without the brackets the preprocessor will expand

```
amount = TAX * 21.50;
```

into

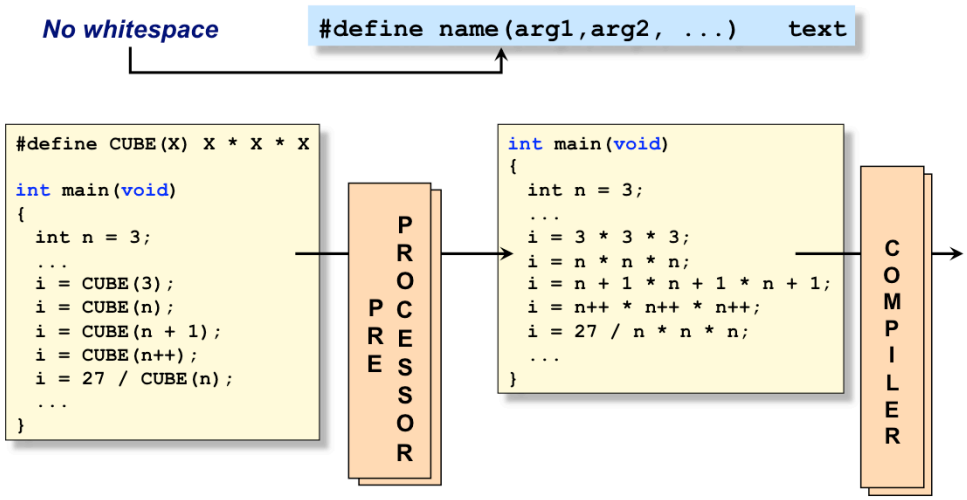
```
amount = 1.5 + VAT * 21.50;
```

and C precedence rules mean that 1.5 is not included in the multiplication. But with brackets

```
#define  TAX  (1.5 + VAT)
```

Function Macro

- #define can take parameters



#defines can take parameters and behave like functions in C. Such functions are called macros. In the example above we should have defined CUBE(X) as

((X) * (X) * (X))

with brackets everywhere. Observe the expansions that occur without the brackets. The last three examples are all incorrect because of precedence rules. Remember, if you use a #define put in brackets everywhere!

Function or Macro

- **Preprocessor runs before the compiler**
  - functions with the same name as a macro will be hidden
- **Preprocessor Macros**
  - are typeless
- **Functions**
  - are typesafe

bracket *fools* the preprocessor

```
#define   CUBE(X) X * X * X
int CUBE(int x);

int main(void)
{
     ...
     result = CUBE(3);
     result = CUBE(53.92);
     result = (CUBE)(3);
     ...
}
```


You can define functions as macros or as true C functions. Macro definitions are typeless, unlike their C counterparts and this means the macro can be called with ints or doubles (the pre-processor treats both as text).

Note that if you define a C function and a macro with the same name, calls such as

result = CUBE(3)

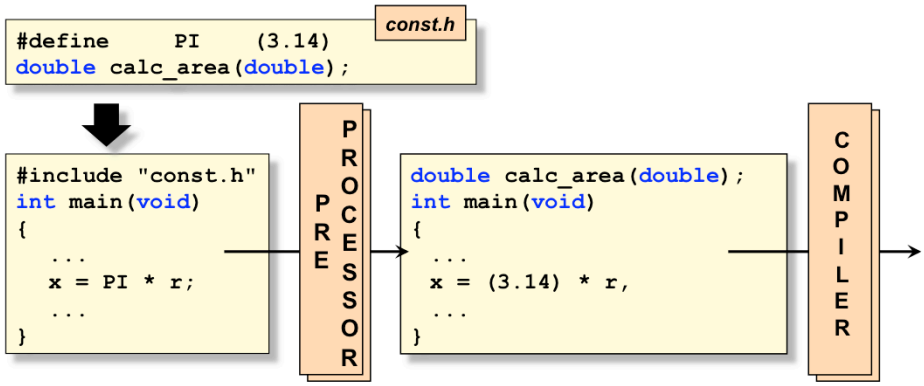
appear ambiguous, but the macros gets called (effectively masking the C function) because the pre-processor runs as the first pass of the compiler. If you really want to call a C function with the same name as a macro, place brackets round the function name to fool the pre-processor (they are ignored by the compiler):

result = (CUBE)(3)

#include

- **Includes the entire contents of the file specified**
  - **<> for system header files**
  - **"" for programmers header files**

```
#include <filename>
#include "filename"
```



The other main uses of the pre-processor is #include. A #include effective pastes the contents of an entire file into your source code. You can use #include in one of two ways. If the filename is enclosed in angle brackets, the search for header file looks through standard directories (as specified by the compiler). If the filename is enclosed in double quotes, the compiler uses the name as an absolute or relative pathname starting from the current directory. If that search fails, the compiler adopts the search rules for the angle-bracket notation.

Header Files

- **Include files have a .h extension**
  - **C++ drops the .h when using libraries and uses <cstring> instead of <string.h> etc**

MyHeader.h

```
#ifndef MYHEADER_H
#define MYHEADER_H
    ...
    #define     CHAR_BIT    (8)
    #define     INT_MAX     (+32767)
    #define     INT_MIN     (-32768)
    #define     SHRT_MAX    (+32767)
    #define     SHRT_MIN    (-32768)
    ...
    void myFunction(int, int);
    ...
#endif
```


A header file is an ordinary text file. It contain text to be #included in your source file. You are advised to adhere to certain rules:

1) do not include function definitions. This may lead to illegal duplicate function definitions and the linker will fail.

2) do not include global data definitions. This will lead to separate global definitions in each compilation module, i.e. multiple definitions.

Header files can be used to contain #defines and function prototypes. You can also nest #includes.

As a precaution against multiple inclusion with nested #includes, you are advised to wrap all header files with

#ifndef unique-identifier

#define unique-identifier

content-of-header-file

#endif

This will ensure the definitions are only defined once per compilation module, even if the header file is include twice. The first time the header file is called, the unique-identifier will be undefined and therefore the next statement will define it and all the

#if

- This directive allows lines of source text to be conditionally included or excluded from compilation
- General form is...

```
#if expr1
     group_of_lines_1
#elif expr2
     group_of_lines_2
#else
     group_of_lines_3
#endif
```

Only one of these groups is sent to the compiler

- Useful for isolating code dependent on client, OS, version, etc.

The #if construct can be used for conditional compilation. Note that the pre-processor uses syntax that is slightly different from that used by the C language itself.

Advanced Facilities

- **Compiler concatenates adjacent string literals...**

```
"Hello"  " World"  ⟶  "Hello World"
```

- **Text preceded by # is made into a string literal...**

```
#define MAKE_STRING(s)   #s
MAKE_STRING(x > 0)  ⟶  "x > 0"
```

- **Several identifiers are predefined ...**
  - __LINE__ current source line number
  - __FILE__ string literal - name of file

Another feature of the pre-processor is to splice together adjacent string constants. "Hello" " World" will be joined to become a single literal "Hello World" (note that the space is a character from the second string). This mechanism is useful when writing long string constants, usually in printf statements.

Creating string literals with the pre-processor requires the use of another feature; he # operator. The # operator creates a string from its single argument.

Several predefined symbols can be used. To avoid conflicts with your symbols, the preprocessor symbols all begin and end with a double underscore.

Stringizing Operator

- **The # operator can be used to convert macro parameters into strings**
  - **encloses parameters in double quotes**

```
#define  VALUE(expr) printf("%s = %i\n", #expr, (expr))
    ...
VALUE (i * j + 5);
```

```
printf("%s = %i\n", "i * j + 5", (i * j + 5));
```


If the # stringizing operator is used before a macro parameter, that parameter is turned into a string (it is surrounded by double quotes). This is handy when writing debug macros such as the example shown above.

#pragma

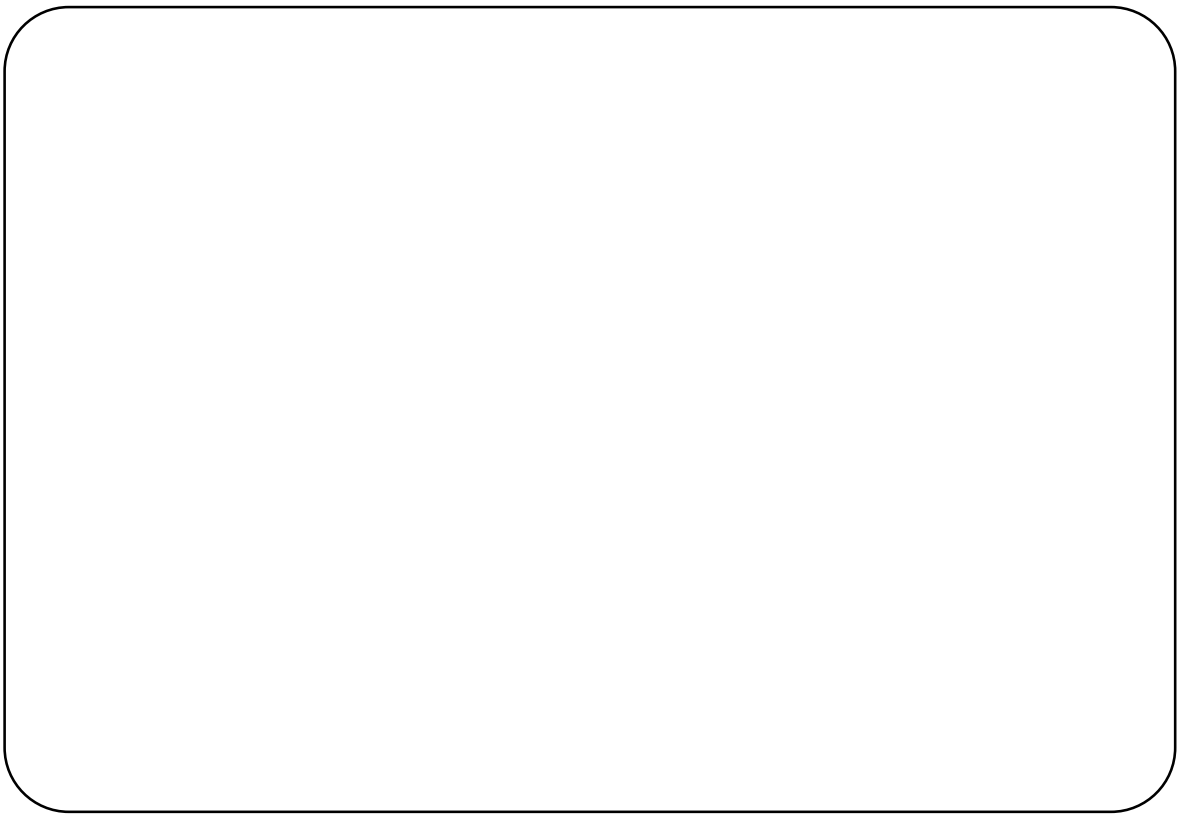
- #pragma allows compiler specific definitions
  - no standard that specifies what pragmas should be supported by C compilers
- Unrecognised pragmas are ignored

```
#pragma check_pointer(off)

for (p = a, end = &a[SIZE]; p < end; p++)
     *p = 0;

#pragma check_pointer(on)
```


The last feature of the pre-processor we will discuss is #pragma. This directive is read by the pre-processor, but affect either the compiler or the optimiser and the way it produces code. Pragmas provide a way of guaranteeing that code is compiled with certain options in force, rather than leaving things up to the user (who may not know what flags to provide to the compiler). There is no standard list of pragmas, each compiler defines it own set of pragmas.

As an example, the following list shows some of the more commonly-used pragmas supported by the Microsoft compiler:

#pragma check_pointer	enable/disable pointer checking
#pragma check_stack	enable/disable stack checking
#pragma function	specifies that calls to a function will take place as a conventional function call
#pragma intrinsic	specifies that calls to a function will be expanded inline
#pragma pack	controls structure packing

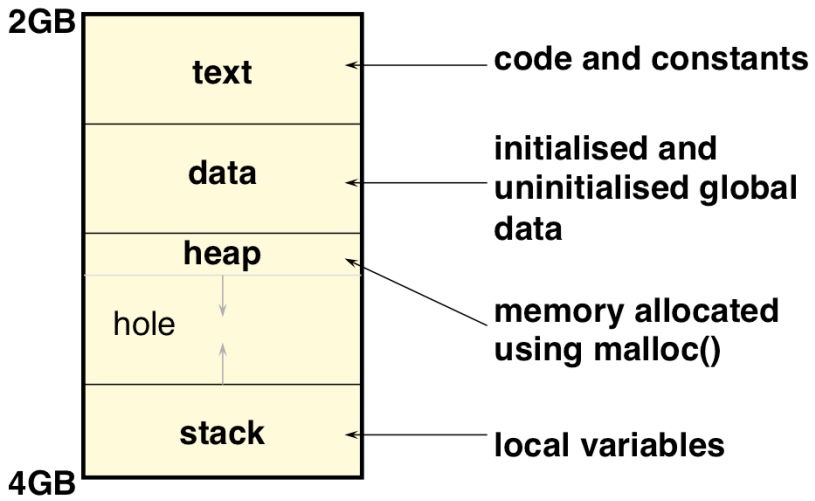


Dynamic Memory Allocation

- **Arrays**
  - relationship to pointers
  - pointer notation
- **Manipulating Pointers**
  - pointers to arrays
  - indexing
  - pointer arithmetic
- **Functions**
  - call by value
  - call by reference

13

Process Memory Model



A process executes in its own virtual memory or virtual address space. The virtual memory is 4GB on 32 bit processors; 2GB is available to the application and 2GB is reserved for the kernel.

The virtual memory of a process is organised into sections or "segments", as shown above. The text portion of a process is the area of memory containing the instructions being executed (code and constants). The virtual-memory system arranges for this memory to be read-only. The data area contains all initialised and uninitialised, static and global variables. The heap area is used when more memory is allocated to the process dynamically (using malloc() and related routines). The stack area is used to hold the stack frames of the process, which may contain return addresses from routines and automatic variables of the routines being called.

There will normally be a (very large) hole in the memory map between the heap and the stack. This region is used for operating system dependent operations, such as shared memory and mapping databases into memory.

This chapter looks at how to exploit the heap region.

Using the Heap/Hole

- The HEAP is used to allocate memory at run time
  - dynamic storage
  - up to 2GB virtual memory available on 32 bit machines
  - main storage area in large programs
  - use memory allocators
    - allocated uninitialised memory `malloc()`
    - allocated initialised memory (to zero) `calloc()`
    - free memory `free()`
- The HOLE is used for shared memory
  - operating system specific calls
  - processes can share memory segments
  - databases can be mapped into shared memory

Most programs need to allocate memory at run-time. The HEAP and the HOLE are used for this purpose. The C runtime library contains routines to utilise the heap; operating system API routines are used to utilise the hole.

The C runtime library provides a standard set of memory allocators to work with the heap:

malloc	allocated uninitialised memory
calloc	allocated initialised memory (to zero)
free	deallocate memory

It is the programmer's responsibility for deallocating any memory previously allocated. Failure to do will result in a memory leak. In practice this means unused segments of memory will clog up the machine's swap area and tend to slow down other applications. This will only be a serious problem on long running applications; memory is deallocated automatically by the kernel when a program finishes.

Use of routines to map databases and shared memory segments is beyond the scope of the course, but in practice is quite simple. The Windows and Unix APIs for the hole have different names, but are very similar (see VirtualAlloc, CreateFileMapping for Windows and mmap, shmat for Unix).

Using the Memory Allocator

- use sizeof
  - to ensure correct amount of storage is allocated
- use typedef
  - with anonymous structure
  - to simplify notation

```
#include <stdio.h>

typedef struct {
    int euros;
    int cents;
} Money;

int main(void)
{
    Money* ptr = malloc(sizeof(Money));
    (*ptr).euros = 150;
    (*ptr).cents = 27;

    // use structure

    free(ptr);
}
```


To allocate memory on the heap we normally use malloc. You can use malloc to allocate a single structure (unusual) or an array of structures on the heap. malloc allocates in units of bytes, so you must calculate the size of the region you want and pass it as a parameter to malloc. malloc will return a pointer to the region allocated (or a NULL pointer on failure - extremely rare). Use the sizeof operator to ensure your calculation is correct.

Once a region on the heap is allocated you can use standard pointer notation to access the region.

Note that the above example could have been written

```
Money* ptr = malloc(sizeof(Money));
ptr->euros = 150;
ptr->cents = 27;
free(ptr);
```

Also note the use of the typedef to avoid having to specify the struct keyword when referring to Money structures. We have defined an anonymous structure an then used the typedef to provide alias to this structure called Money.

Allocating Arrays

- **Arrays of Structures**
  - allocated in same way
  - use array notation

```
#include <stdio.h>

typedef struct {
    int euros;
    int cents;
} Money;
```

```
int main(void)
{
    Money* ptr = malloc(100 * sizeof(Money));

    // use array
    ptr[0].euros = 150;
    ptr[0].cents = 50;
    ptr[99].euros = 152;
    ptr[99].cents = 25;

    // release array
    free(ptr);
}
```


If we allocate arrays of structures, it is convenient to use the pointer-array duality idioms discussed earlier to simplify coding.

Recall that a pointer to a memory region can be considered as the name of an array. Here, the pointer returned by malloc is considered an array. Standard [ ] notation can be used to access components of each structure.

The only difference here is that malloc returns an l-value and technically an array pointer should be an r-value. So be very careful you don't modify the pointer returned by malloc!

Functions Allocating on the Heap

```
Money* AllocateArray(int count)
{
    Money* ptr = malloc(count * sizeof(Money));
    return ptr;
}

int main(void)
{
    Money* salaries = AllocateArray(100);
    salaries[0].euros = 150;
    salaries[0].cents = 50;
    salaries[99].euros = 152;
    salaries[99].cents = 25;

    free(salaries);
}
```

```
#include <stdio.h>

typedef struct {
    int euros;
    int cents;
} Money;
```


The last example can be extended by providing a function to perform the memory allocation. The function can encapsulate the calculation of the size allocated and return the correct pointer type.

Linked Lists

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

struct Money {
    struct Money* next;
    int euros;
    int cents;
};

int main(void)
{
    struct Money* salary1 = CreateMoney(150, 50);
    struct Money* salary2 = CreateMoney(130, 20);
    struct Money* salary3 = CreateMoney( 60, 00);
    salary1->next = salary2;
    salary2->next = salary3;
}
```

```
struct Money*
    CreateMoney(int euros, int cents)
{
    struct Money* ptr
        = malloc(sizeof(struct Money));
    ptr->euros = euros;
    ptr->cents = cents;
    ptr->next = NULL;
    return ptr;
}
```


In this example we have a function that allocates a single Money structure and initializes it with the two input parameters. This mirrors the way a constructor is used in C++, Java and C#.

The structures returned can be added to a linked list using an additional pointer, next. Linked lists are common in C programming; in practice they tend to be somewhat more sophisticated than this example, but the example shows the general idea.

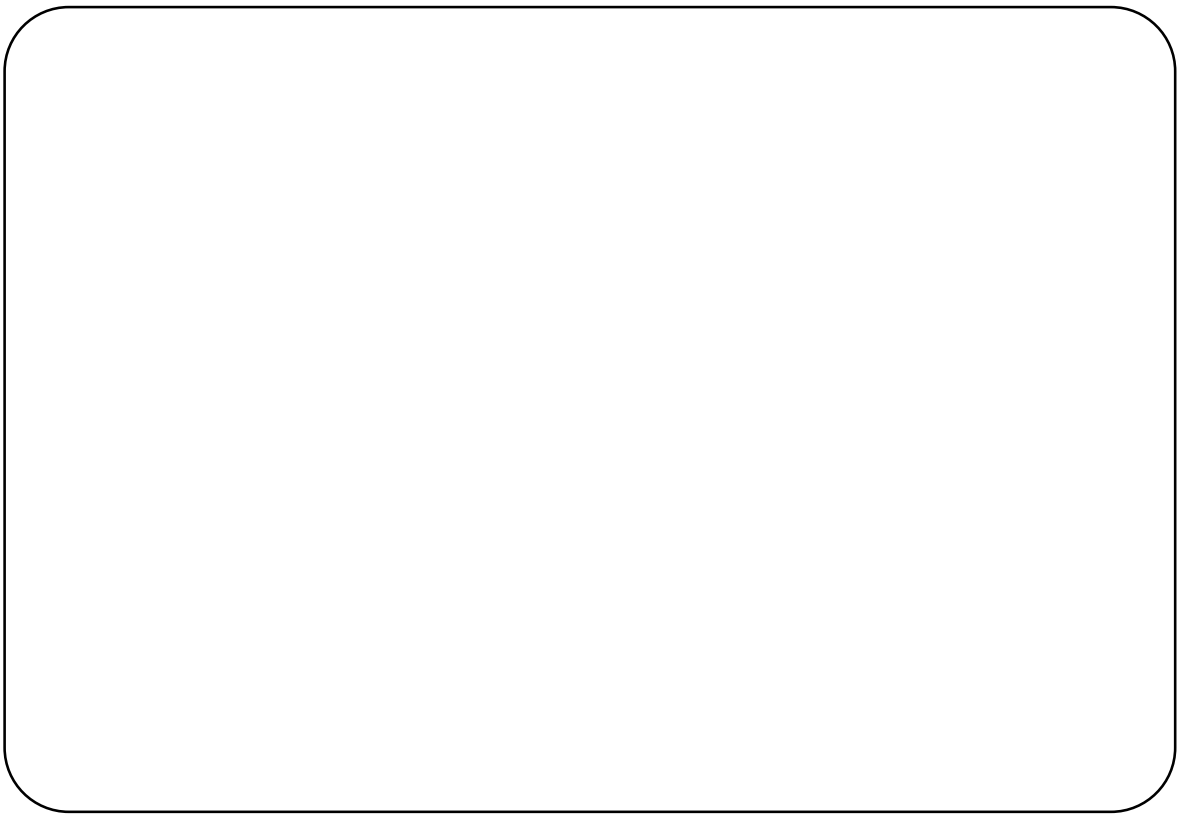
Other complex structures can be defined along similar lines. Unfortunately, there is no support in the C runtime library for complex structures - you have to write your own. The situation in C++ is completely the reverse - there are all sorts of containers defined for these complex structures. It is probably worth using the C++ libraries rather than try writing your own.

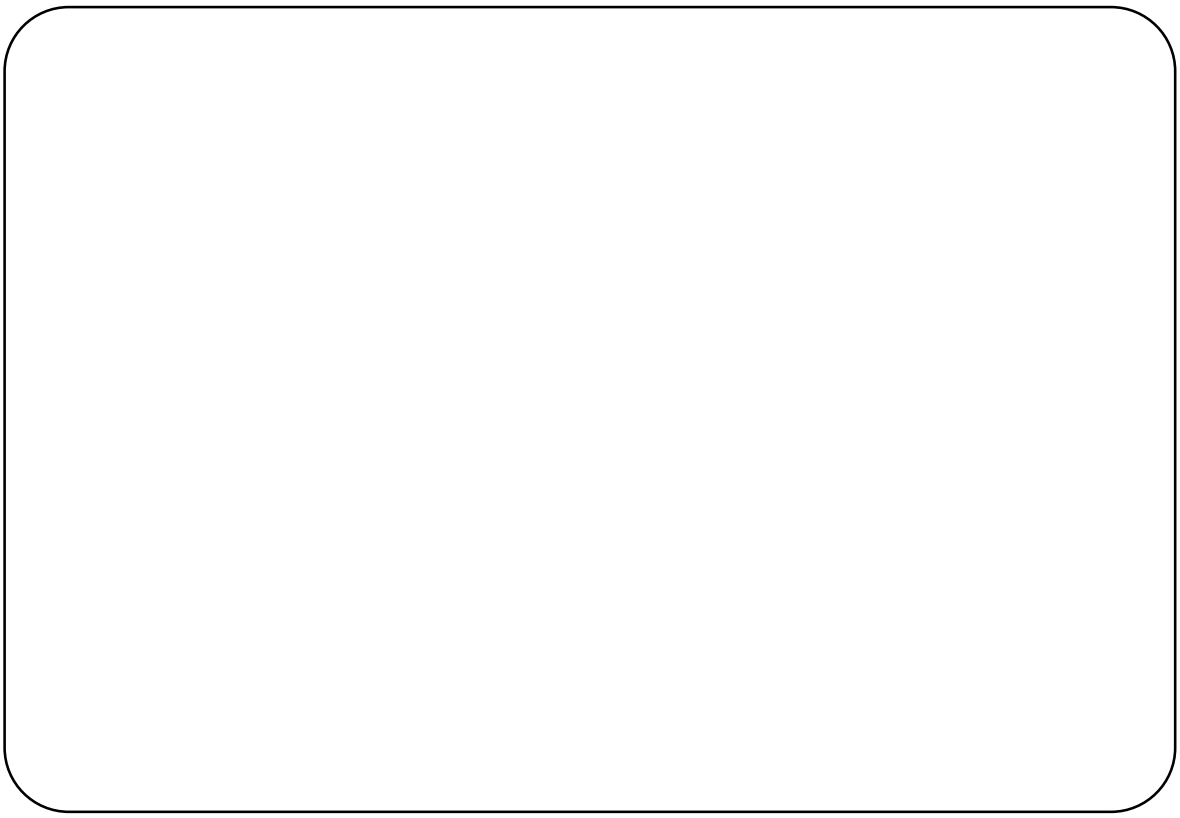
Linked Lists with typedef

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

struct Money {
    struct Money* next;
    int euros;
    int cents;
};

int main(void)
{
    Money* salary1 = CreateMoney(150, 50);
    Money* salary2 = CreateMoney(130, 20);
    Money* salary3 = CreateMoney( 60, 00);
    salary1->next = salary2;
    salary2->next = salary3;
}
```

```
Money* CreateMoney(int euros, int cents)
{
    Money* ptr = malloc(sizeof(Money));
    ptr->euros = euros;
    ptr->cents = cents;
    ptr->next = NULL;
    return ptr;
}
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.





gcc Compiler

- **gcc is really a compiler and a linker**
  - **can compile multiple files in one go**
  - **or one at a time (using –c)**

```
gcc  [-c]           // compile only
     [-std=C11]     // choose standard
     [-g]           // debug symbols
     [-O3]          // optimization level
     [-Wall]        // all warning
     [-Wpedantic]   // strict compliance
     [-Idir...]     // include directories
     [-Ldir...]     // library directories
     [-Ddebug=1]    // macro symbols
     [-o outfile]   // output file
     file1.c ...    // input files
```

```
$ ls f*.c
f1.c  f2.c  f3.c  f_main.c
$ gcc -g -Wall –Wpedantic f1.c f2.c f3.c -o f.exe
$ ls f*
f1.c  f1.o  f2.c  f3.c  f.exe  f.h  f_main.c
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

Makefiles

- **Recipe for building executables**

```
$ ls f*.c
f1.c  f2.c  f3.c  f_main.c
$ make -f Makefile2
gcc -g -O0 -Wall -std=c11  f1.c -c
gcc -g -O0 -Wall -std=c11  f2.c -c
gcc -g -O0 -Wall -std=c11  f_main.c -c
gcc -g -O0 -Wall -std=c11  f3.c -c
gcc -g -O0 -Wall -std=c11  f1.o f2.o f_main.o
```

```
CC := gcc
WARNINGS := -Wall
CFLAGS := -g -O0 $(WARNINGS) -std=c11
SOURCES := $(wildcard f*.c)
OBJECTS := $(SOURCES:.c=.o)

all: f.exe
f.exe: $(OBJECTS)
        $(CC) $(CFLAGS) $? -o $@

$(OBJECTS): f%.o: f%.c
        $(CC) $(CFLAGS) $? -c
clean:
        -rm -f *.o
        -rm -f *.exe
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

Static Linking

- **Create a library with ar utility**
  - **can add multiple .o files to archive**
  - **locate the archive using**
    - **-L to specify directory containing shared object**
    - **-l to specify name of libary**
      - **assumes libxx.a**

```
cd mylib
gcc -c f1.c
ar -cvr libf1.a f1.o
ar -t libf1.a

cd ../src
gcc f_main.c f2.c f3.c -L../mylib -lf1 -o f.exe
f.exe
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

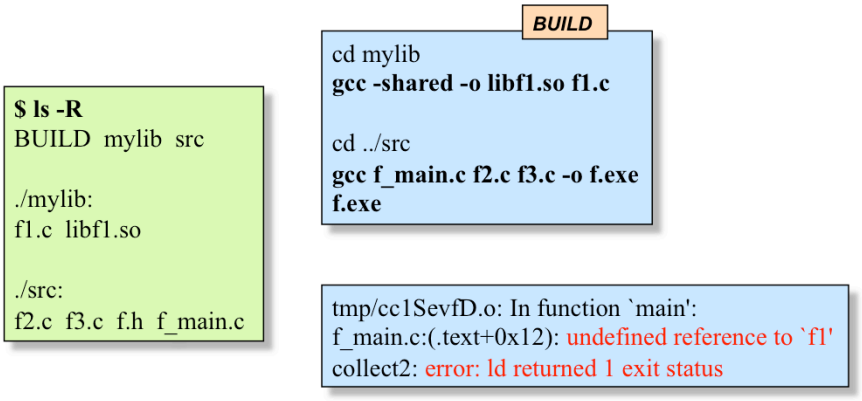
Dynamic Linking

- **Static Linking**
  - Exe is complete
  - no additional code is needed at runtime
  - larger executables
  - nothing is shared
- **Shared Objects (DLL)**
  - Exe is incomplete
  - additional code is needed at runtime
    - e.g. printf
  - smaller executables
  - nothing is shared

Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

Shared Objects ...

- **Create a shared object as a library**
  - **but when we build we can't locate the library**



Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

... Shared Objects

- **Shared Object can be located in several ways**
  - **LD_LIBRARY_PATH**
    - **relies on user setting this environment variable**
  - **using rpath**
    - **executable contains a relative path to *"so"* file**
  - **placing the library in standard directories**
    - **requires root priviledge**
    - **/lib, /usr/lib**
  - **restrictions apply to setuid (setgid) programs**

Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

LD_LIBRARY_PATH

- **Create shared object as usual**
- **Compile code with linker options**
  - **-L to specify directory containing shared object**
  - **-l to specify name of libary**
    - **assumes libxx.so**

```
cd mylib
gcc -shared -o libf1.so f1.c

cd ../src
export LD_LIBRARY_PATH=../mylib
gcc f_main.c f2.c f3.c -L../mylib -lf1 -o f.exe
f.exe
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

rpath

- **Create shared object as usual**
- **Compile code with linker options to specify relative path of shared object**
  - **-Wl,... to pass option on to linker**
  - **-rpath=... directory containing shared object**

```
cd mylib
gcc -shared -o libf1.so f1.c

cd ../src
gcc f_main.c f2.c f3.c -Wl,-rpath=../mylib -L../mylib -lf1 -o f.exe
f.exe
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

Creating a System Library

- **Need root priviledges**
  - **only need to specify –l option**

```
cd mylib
gcc -shared -o libf1.so f1.c
sudo cp libf1.so /usr/lib
sudo chmod +x /usr/lib/libf1.so

sudo ldconfig   # update cache
sudo ldconfig -p | grep libf1.so

cd ../src
gcc f_main.c f2.c f3.c -lf1 -o f.exe
f.exe
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

Displaying Symbol Table

- **list symbols in object/exe file**

```
$ cat myfile.c
int x = 100;
double y = 100.57;
struct Point
{
   int x;
   int y;
};
struct Point myPoint;
void move(struct Point* p)
{
   p->x = 10;
   p->y = 10;
}

$ gcc -c myfile.c
```

```
$ nm myfile.o
00000000 T move
00000008 C myPoint
00000000 D x
00000008 D y
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

cachegrind

- ar archive utility used to build static libraries

Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

ranlib

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

struct Money {
    struct Money* next;
    int euros;
    int cents;
};

int main(void)
{
    Money* salary1 = CreateMoney(150, 50);
    Money* salary2 = CreateMoney(130, 20);
    Money* salary3 = CreateMoney( 60, 00);
    salary1->next = salary2;
    salary2->next = salary3;
}
```

```
Money* CreateMoney(int euros, int cents)
{
    Money* ptr = malloc(sizeof(Money));
    ptr->euros = euros;
    ptr->cents = cents;
    ptr->next = NULL;
    return ptr;
}
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

valgrind

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

struct Money {
    struct Money* next;
    int euros;
    int cents;
};

int main(void)
{
    Money* salary1 = CreateMoney(150, 50);
    Money* salary2 = CreateMoney(130, 20);
    Money* salary3 = CreateMoney( 60, 00);
    salary1->next = salary2;
    salary2->next = salary3;
}
```

```
Money* CreateMoney(int euros, int cents)
{
    Money* ptr = malloc(sizeof(Money));
    ptr->euros = euros;
    ptr->cents = cents;
    ptr->next = NULL;
    return ptr;
}
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

cachegrind

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

struct Money {
    struct Money* next;
    int euros;
    int cents;
};

int main(void)
{
    Money* salary1 = CreateMoney(150, 50);
    Money* salary2 = CreateMoney(130, 20);
    Money* salary3 = CreateMoney( 60, 00);
    salary1->next = salary2;
    salary2->next = salary3;
}
```

```
Money* CreateMoney(int euros, int cents)
{
    Money* ptr = malloc(sizeof(Money));
    ptr->euros = euros;
    ptr->cents = cents;
    ptr->next = NULL;
    return ptr;
}
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.

callgrind

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

struct Money {
    struct Money* next;
    int euros;
    int cents;
};

int main(void)
{
    Money* salary1 = CreateMoney(150, 50);
    Money* salary2 = CreateMoney(130, 20);
    Money* salary3 = CreateMoney( 60, 00);
    salary1->next = salary2;
    salary2->next = salary3;
}
```

```
Money* CreateMoney(int euros, int cents)
{
    Money* ptr = malloc(sizeof(Money));
    ptr->euros = euros;
    ptr->cents = cents;
    ptr->next = NULL;
    return ptr;
}
```


Finally, the same example, but this time with the typedef. The typedef simplifies notation even in a simple example like this. In more complex situations you should always strongly recommended to use a typedef.