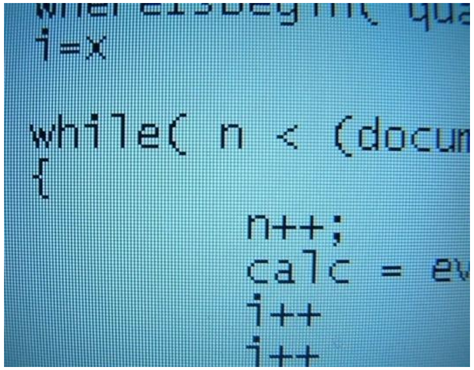
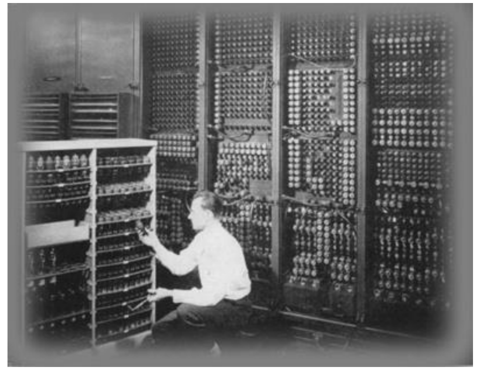




CS1PR16

Advanced Type Features

Learning Objectives

- Define constants and list their benefits
- List the scope (visibility) of identifiers
- Define the rules for data conversion
- Define the syntax and semantics of derived datatypes: struct, enum, union and bit fields
- Utilise derived datatypes to design more complex programs
- Utilise function pointers to create a flexible function

Outline

- Advanced types
  - Modifiers
  - Constants
  - Type conversion
- Compound data types
  - Array
  - Struct, enum, union
  - Bit fields
- Function types

Variable Modifiers

- Basic types (int, char, ...) may have their characteristics modified
  - Syntax: <modifier> <type> <variable>
  - Example: `short signed int my_int;`
  - Modifiers:
    - `short (makes the length shorter, fewer bits)`
    - `long (makes it longer, you can use long long)`
    - `signed (have a sign)`
    - `unsigned (don't have a sign)`
- Short and long modifies the size of the variable
  - The actual length is system-specific
  - E.g., better performance to compute 16-bits on a 16-Bit CPU
  - Doesn't matter nowadays!
- Avoid these types, use the types with fixed length (e.g. int32_t)
- You will see these types in existing code, though!

Constants

- Constants are variables that cannot change
- Declaration: Syntax

```
<type> const <identifier> = <expression>;
Example:
  int const numPlanets = 8+1;
```

- Read from left to right: *numPlanets is a constant integer*
- The compiler will check that it isn't changed
  - Produces a warning or error (depending on setting)
- Some programmers use all CAPITALS for constants

Constants versus Literals

- The benefit of constants:
  - Why not just use a variable without const?
    - A programmer might accidentally change it
    - It is useful for functions, to clarify the meaning of an argument
  - Why not just use a literal (or expression), e.g., 5?
    - A variable allows the program to maintain the meaning
    - Enables to change the variable once and use everywhere

Determining the Size of Types

- `sizeof` is an in-built C function that returns the number of bytes of memory that are used to store a type

```
printf("sizeof int32_t %d\n", sizeof(int32_t));
printf("sizeof short int %d\n", sizeof(short int));
printf("sizeof float %d\n", sizeof(float));
printf("sizeof double %d\n", sizeof(double));
```

- It can be used on types or on variables

```
short int pea;
printf("sizeof variable pea is %d\n", sizeof(pea));
```

Type Casting

- **Value: precise meaning of the contents of an object when interpreted as having a specific type [ISO/IEC 9899]**
- **Type casting: Converts the value of a variable between types**
  - The compiler ensures that the value of the data is converted
    - To the correct data representation (int vs. uint vs. floating-point)
  - This may imply to change the data representation
- Reason: to match a (function) type, to preserve memory

- **Implicit type conversion** is done automatically by the compiler
  - E.g., changing the value of an expression from integer to float

```
int a = 5;
float b = a * 0.5;
```

  - Conversions preserve the value but may loose some precision (int -> float)
    - A value-preserving conversion does not lose precision, e.g., int32_t to int64_t

Type Casting

- **Explicit conversion by the programmer between any type possible**
  - Notation: (<Type>) <Variable>

```
float b = 3;
int a = (int) b;
```

- Beware of programming errors!
  - Example with correct syntax but unexpected error (crash):

```
FILE * p = (FILE*) 4712;
fgetc(p)
```

Casting Example

- What is the output?

```
#include <stdio.h>

int main()
{
  int   i=3;
  char  j='1';
  float my_pi=3.1415;

  printf("int i is:%d \n" , i);
  printf("char j is: %c \n", j);
  i=(int)(j);  // cast j to an int
  printf("j casted to int is:%d \n", i);
  i=(int)(my_pi);     // cast my pi to an int
  printf("my_pi casted to int is:%d \n", i);
  return 0;
}
```

Aggregate Types

- Construct more complex types from basic types
  - They cannot be compared, i.e., “x == y” is not valid
  - Sometimes they cannot be assigned, i.e., copying data between objects
- Array: contains multiple elements of one type
- Struct: contains multiple members of different type
  - Collections of related variables (aggregates) under one type
  - Can contain variables of different data types
- Union: select ONE of the contained members
  - Contains only ONE
- Enum
  - List of identifiers that represent an integer

Arrays

- An array is an object that can store n-items of a data type
  - An array is an **aggregate type**
- Syntax: <declaration>[<*number*>];
  - Where *number* is the number of elements the array shall have
  - The number must be known when the array is defined
  - Example: `int temp[3];`
- Access to individual array elements with square brackets

  `temp[0]` This specifies the first element of the array

  - This expression can be an lvalue or an rvalue!
  - You can also use an integer to specify the position: `temp[x]`
- Arrays cannot be assigned as a whole:

  temp = temp2; // Error: assignment to expression with array type
- Important: Always make sure arrays are initialised

Initialising Arrays

- There are many ways to initialise data
- Using a loop:

```
int temp[3];
for(int i=0; i<3; i++){
      temp[i]=i+1;
}
```

- Directly initialise an array by assigning {} in the declaration

```
int temp[3] = {1, 2, 3};
```

- Using the `memset()` function
  - memset(<memory>, <character>, <size>)
  - Sets each byte in the memory (of size) to the specified character

```
int temp[3];
memset(temp, 0, sizeof(temp));
```

Array Magic!

- With C99, you can define the array size based on a variable:

```
int data[<expression>];
```

The expression will be evaluated and the array size will be fixed at this line!

- Initialise based on known data (implicit size)

```
int data[] = {1,2,3,4,5};
```

- When used with const, the members cannot be changed

```
int const data[] = {1,2,3,4,5};
data[3] = 3; // error: assignment of read-only
```

- More than one index represent multiple dimensions [x][y]
  - Useful for tables (2D) or even images
  - Example of a 2D image:

```
int image[640][480];
image[0][0]=255;
image[639][479] = 255;
```

Arrays

Good practice: define a constant for the size of arrays

```
#define SIZE 10 /* C preprocessor macro */

int x[SIZE];
int i, j;

for(i=0; i<SIZE; i++){
  x[i] = ...
}
...
for(j=0; j<SIZE; j++){
  ...x[j]...
}
```

Wrong code to access invalid members (e.g.)

```
x[SIZE]
x[SIZE+1]
x[-1]
```

Certainly the program is wrong!

- **You are really lucky when it crashes!**
- **In the bad case, you try to debug a ghost**

Your responsibility to enforce arrays bounds

Using Arrays and Casting

We can use casting and arrays to understand how bits are stored

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
int main(){
  int32_t var = 1024 + 64 + 32 + 4; // = 1124
  uint8_t b[4];
  // The following function copies data (here 4
  // bytes) from one memory "location" (var) to b.
  memcpy(b, &var, sizeof(var));

  for(int i=0; i < sizeof(i); i++){
       printf("%d ", b[i]);
  }
  printf("\n");
  return 0;
}
```

- This code prints the four bytes of the integer: 100 4 0 0
  = 100 + 4*256 (second byte). The order is machine specific!
- *Details about the & (memory location) notation soon*

Strings

- Remember, we defined a string as a `char*`
- A string is actually an array of characters (chars) followed by a termination character (ASCII: 0 or \0)
- Use single quotes ‘ for single chars and double “ for strings
  - ‘h’ ‘e’ ‘l’ ‘l’ ‘o’ <= each is a single character
  - “hello” <= array of characters, terminates with \0
- Strings in C end with the Null Character ‘\0’
  - known as Null Terminated Strings
  - automatically used with the “ ” notation
  - *You do not know the size of a string until you find the character \0!*

Strings

University of Reading

```
#include <stdio.h>

int main(){
  char mood[4]={'f', 'u', 'n', '\0'};
  char mod2[] ={'f', 'u', 'n', '\0'};// equivalent

  char data[] = "fun"; // equivalent, readability!

  printf("This is %s == %s\n", mood, data);
  return 0;
}
```

- Use the array notation {} for strings only if you need special ASCII characters

Structures

- A structure contains members (types with names)
  - A `struct` **cannot** contain an instance of itself, but other structures
  - A structure declaration **does not** reserve space in memory (just a type)
- The declaration syntax is as follows:

```
struct <identifier>{
   {<type> <identifier>;}
}; // NOTE THE ; at the end of a struct
```

- Example:

```
struct person{
      int age;
      char name[32];
};
```

  - **person** is the structure *type* and is used to declare variables of this type
  - **person** contains two *members* of type **int** and **char[]**
    - The identifiers of these members are age and name

Using the Structure Type

- Declaring new variables
  - Declared similarly to other variables **after** struct previously defined:

```
struct person jack;
struct person friends[100];
```

  - Can use a comma-separated list, directly after the type declaration:

data type

```
struct person {
    int age;
    char name[32];
} jack, friends[100];
```

variable identifier an array of 100 structures

Structure Operations

- Assigning a structure variable to a structure variable of the same type

```
person1 = person2;
```

- Accessing the members of a variable using dot punctuator "."

```
theAge = jack.age;
printf("name is: %s\n", jack.name);
```

- Using the sizeof operator to determine the size of a structure

```
size = sizeof(jack);
```

- Using the offsetof operator to determine the location of a member
  - Offset inside the structure (we will see this!)

  Syntax: offsetof(<struct|union type>, <member>)

  Requires: `#include <stddef.h>`

```
pos = offsetof(jack, age);
```

- Determining the memory address of a structure variable

```
ptr = & jack;
```

Initialising Structures

- Assignment statements
  - Directly assign one struct to another

    ```
    struct person author = jack;
    ```
  - Define and initialise one member at a time

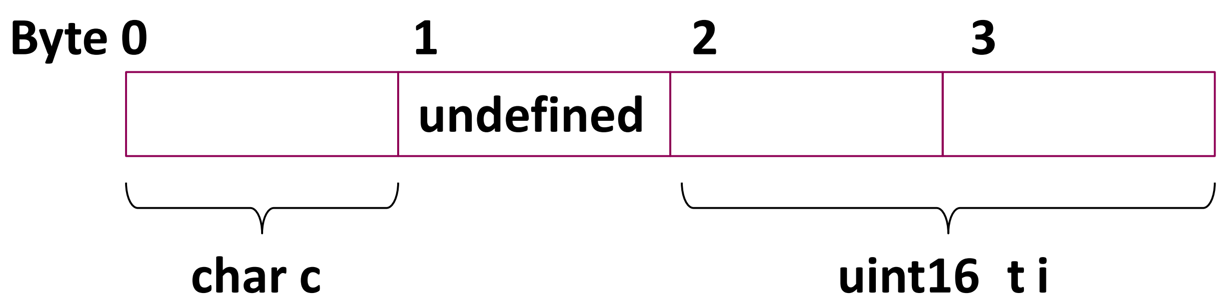
    ```
    struct person author;
    author.age = 87;
    author.name = “a name...”;//Cannot assign an array, though!
    ```
- Initialiser list: set all the members in one statement
  - Put the values in a comma separated list inside { }
  - Example: `struct person jack = { 20, “Jack Kerouac” };`
  - Warning: if you change/reorder the structure members, this is a mess!
- Initialiser list with fieldnames
  - Fields can be assigned using .<field> = <expression>
  - Allows to reorder/change structures => may lead to compiler errors!
  - Example: `struct person jack = { .age = 20, .name = "Jack" };`

Structures in memory

- The structures must be represented in memory
- The compiler aligns the members for efficiency
  - e.g. integers are aligned to 4-byte boundaries
  - Data type and architecture/instruction-set specifics
  - For efficiency: group the same types, first big types

```c
struct example_t {
   char c;
   uint16_t i;
};
```

Memory representation



- There are other possible mappings, this is system-specific
  - The offsetof() function allows to identify the offset of a member to begin

Group Work

Task:

1. Write down a structure for information contained on a DVD
   - At least four members

2. Sketch a possible representation of the memory
   - (If you miss information, make a guess)

Time: 3 min

```
struct example_t {
   char c;
   uint16_t i;
};
```

Typedef

- Typedef
  - Creates synonyms (aliases) for previously defined data types
  - Useful for complex types to create shorter memoizable names
  - Syntax: typedef <known type> <new type>
  - Example:

```
typedef struct card CardType_t;
CardType_t card1;
```

  - Typedef does not create a new data type: it only creates an alias
- Good practice: use the suffix _t for non-trivial types
- Can also be used for standard types:

```
typedef _Bool bool;
```

- Allows changing data types (and their names)
  - without changing all code
  - Consider using int32_t for math by default or float

Unions

- Data type that contains a variety of objects over time
  - Only contains one data member at a time
  - Members of a union share memory space
    - Conserves storage
  - Programmers must take care to only access the correct element
    - Often done by storing a union in a struct together with the type
- Union declaration
  - Same as struct with members

```
union myu {
  int16_t x;
  float y;
};

union myu var;
```

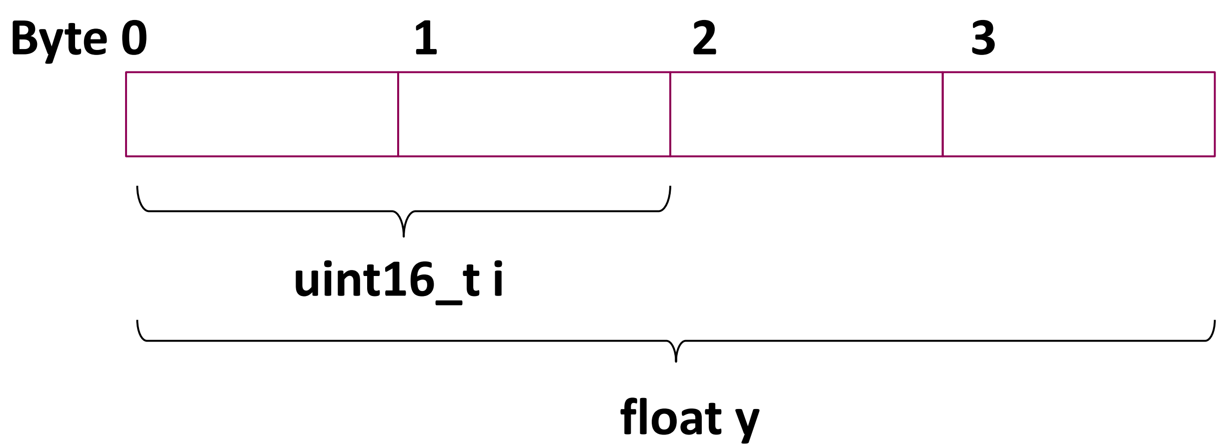
- Access to a member using the dot "." punctuator again

Unions in Memory

- Members of a union share memory space
  - Take the biggest size
  - If one type is bigger than another, fill it partially
- That is why you need to know what type is stored!

```
union myu {
  int16_t x;
  float  y;
};
```

Memory representation



Group Work

Task:

- How can we use a UNION to access the bytes in an integer?
  - Without using memcpy()!

Time: 3 min

Share: 1 min

Our previous code that performs this job was a bit bulky

```
int main(){
  int32_t var=1024 + 64 + 32 + 4;
  uint8_t b[4];
  // The following function copies data (here 4
  // bytes) from one memory “location” (var) to b.
  memcpy(b, & var, sizeof(var));

  for(int i=0; i < sizeof(i); i++){
        printf("%d ", b[i]);
  }
  printf("\n");
  return 0;
}
```

Group Work: Solution

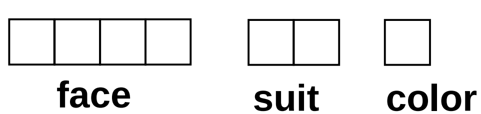
```
union data_t{
  int32_t var;
  uint8_t b[4];
};

typedef union data_t data_t;

int main(){
  data_t v;
  v.var = 1024+64+32+4; // here we store into the union

  for(int i=0; i < sizeof(v.var); i++){
       printf("%d ", v.b[i]);
  }
  printf("\n");
  return 0;
}
```

- Maybe more elegant?

Bit Fields

- A bit field defines the exact size of the variable in bits
  - Typically, a member of a structure
  - Enables better memory utilisation, for networking protocols
  - Must be defined as int or unsigned
  - Programmers cannot access individual bits directly (need bit-ops)
- Defining bit fields
  - Follow unsigned or int member with a colon (:) and an integer constant representing the width of the variable
  - Example:

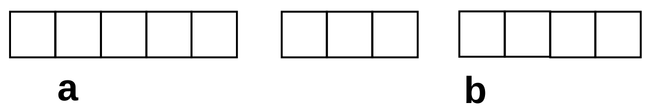
```
struct bitCard_t {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};
```



Bit Fields

- Unnamed bit field are used as padding in the structure
  - Nothing should be stored in the bits
  - Useful for compatibility with some binary format / exchange protocols

```
struct Example {
    unsigned a : 5;
    unsigned   : 3;
    unsigned b : 4;
}
```



  - Unnamed bit field with zero width aligns next bit field
    - to a new storage unit boundary
    - Data at a storage unit boundary can be accessed efficiently

Enumerators

- enum declares a list of constants (integer) values
  - by default starting at zero, and each successive element is increased in one

```
enum <identifier> {<identifier>, <identifier>, …};
```

- Example

```
enum boolean {FALSE, TRUE};
printf("value of False is: %d\n", FALSE);
printf("value of True is: %d\n", TRUE);
```

  - Output will be:

```
value of False is: 0
value of True is: 1
```

- The value of the elements can also be given

```
enum day {MONDAY = 1, TUESDAY = 2};
```

- Note that the values must be known and will be replaced at compile time

Enumerators

- Example code: What will be printed?

```
enum months_t{JAN=1,FEB,MAR,APR,MAY,JUN,
                JUL,AUG,SEP,OCT,NOV,DEC};

printf("value of Feb is: %d\n", FEB);
printf("value of Nov is: %d\n", NOV);
```

- An enumerator type can be used as a function argument
  – Supports a clear interface

```
int func(enum months_t);
```

Function Type

- A **function type** is a variable that “references/points” to a function
- Reason: modular programming
    - Some other code decides about the function to use, when we call a function
- Notation for declaration/definition of variables:
    - <return-type> (*<VAR>) (<arguments of the prototype>)

```
int (*funcP)(int, int);
```

- funcP is now a pointer variable for a function with:
    - return type: int
    - two arguments of type: int

- Notation for calling a function using a function type:
- Similar to normal function: (*<VAR>)(<arguments>);

```
ret = (*funcP)(4, 3);
// normal function call: ret = funcP(4,3)
```

Function Type: Example

```
#include <stdio.h>

int squareFunc(double val){
  return (int) (val*val);
}

int main(){
  // declaration of the variable f_ptr as a function pointer
  // expected prototype of the function is: int()(double)
  int (*f_ptr)(double);
  // assigning a function to a function pointer
  f_ptr = squareFunc;

  // calling a function in the function pointer
  int ret = (*f_ptr)(3.4);

  // print return value which is floor(3.4*3.4) ~= 11
  printf("Calling returns: %d\n", ret);
  return 0;
}
```

Function Pointer

- The declaration of variables is non-easy to read:

```
int (*f_ptr)(double);
```

- Typedef improves readability:

```
typedef int (*AnyCoolFunc)(double);

AnyCoolFunc f_ptr = &squareFunc;
```

- A structure may include function pointers, too

```
struct dataSet{
  AnyCoolFunc myFunc;
  int a;
};
```

Function Pointer: Syntactic Sugar

- Actually, the compiler knows we deal with FPs
  - No need to de-reference FP

```
int main(){
 int (*f_ptr)(double);
 f_ptr = squareFunc;
 int ret = f_ptr(3.4);
```

- Advise: Use the notation (*f_ptr)
- Be warned, a FP might be **not assigned == NULL**
  - Causing crashes of the application!
  - Sane programs check function pointers

Summary

- Constants prevent accidental modification
- Typecasting converts the value of a variable to another type
  - Implicit => done by the compiler, between compatibile types
  - Explicit => done by the programmer, flexible, may
- Array: n-elements of the same type
- Structure: named members of possible different types
- Union: memory sharing of members
- Bit field: named groups of bits
- Enumeration: named numbers / constants
- Function pointer: modular way to store/use a function