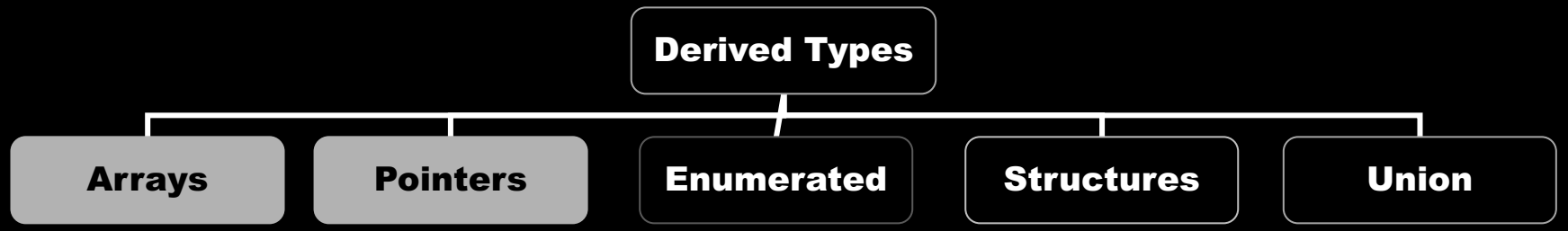
Derived Types in C++



Enumerated Types in C++

- Define enumeration constant - a list of constant integer values

```
// defining enumerated tag and the variables separately
enum StatusType {eSingle, eMarried, eDivorced, eWindowed};
StatusType Status1, Status2, Status3;
// assigning values
Status1 = eMarried;
Status3 = eSingle;
```

```
// defining the tag and declaring the vars simutaneously
enum StatusType {eSingle, eMarried, eDivorced, eWindowed}
                                                StatusA,StatusB;
// assigning values
StatusA = eMarried;
StatusB = eSingle;
```

Enumerated Types in C++

```
enum GoodJobType {eTinker, eTailor, eSoldier, eSpy};

GoodJobType Job1, Job2, Job3, Job4;
Job1 = eTinker;
Job2 = eTaylor;
Job3 = eSoldier;
Job4 = eSpy;

cout  << "tinker = "  << int(Job1) << endl
      << "taylor = "  << int(Job2) << endl
      << "soldire = " << int(Job3) << endl
      << "spy = "     << int(Job4) << endl;
```

```
tinker = 0
tailor = 1
soldier = 2
spy = 3
```

Enumerated Types in C++

```
enum GoodJobType {eTinker, eTailor, eSoldier=9, eSpy};

GoodJobType Job1, Job2, Job3, Job4;
Job1 = eTinker;
Job2 = eTaylor;
Job3 = eSoldier;
Job4 = eSpy;

cout  << "tinker = "  << int(Job1) << endl
      << "taylor = "  << int(Job2) << endl
      << "soldire = " << int(Job3) << endl
      << "spy = "     << int(Job4) << endl;
```

```
tinker = 0
tailor = 1
soldier = 9
spy = 10
```

Enumerated Types in C++

- all `enums` in C are of type int.
- You often find C programs using `enums` to declare sets of integer constants.
- However, in C++ enumerated sets are types with no explicit relation to integers.
- Typecasting is required to convert enums to ints
- `int MyNum = int(MyEnum);`

Structures in C/C++

- structs are aggregate data types.
- A struct is a collection of named members (sometimes referred to as fields).
- Each member can be a different type.

- You can think of a structure as a collection of various data types all referenced by names.

Structures in C++

- In C++ when you wish to create a structure you did the following:

```
struct struct_name
{
      members …
};
```

- When we specify the form of a *struct* in C++ we are creating a type.

Structures in C++

- For example:

```
struct DetailsType {
        int age;
        char address[128];
        char name[64];
};
```

- means
  - a type "DetailsType" is created
  - You can use "DetailsType" to declare variables

```
DetailsType John, Peter, Andrew;
```

Structures in C++

- In C++ *structs* can have data members, member functions, constructors and a destructor.
- In C *structs* can not have member functions, constructors or destructors. Again another difference between C and C++.

Structures in C++

- Example:

```
struct DetailsType{
        int Age;
        char Address[128];
        char Name[64];
};

DetailsType MyDetails;
```

Structures in C++

```
struct SoldierType {
  char Name[20];
  char Rank[50];
  int SerialNo;
};
SoldierType Soldier1;
SoldierType Soldier2={"J.Smith","Private",777};
Soldier1 = Soldier2;
strcpy(Soldier1.Name,"J.Murphy");
cout << Soldier1.Name << endl;
cout << Soldier1.Rank << endl;
cout << Soldier1.SerialNo << endl;
```

```
J.Murphy
Private
777
```

Unions in C/C++

- C++ built in types by definition can hold one value of that type. Eg int, long double.
- In C++ structs can contain several types and even functions.
- A union is a little different. A union can hold the values of several types, however, it can only use one value at a time.

Unions in C/C++

- A unions declaration is similar to a `struct`.

```
union S
{
        int x;
        double y
};
```

Just like a struct, the union above defines a template (tag) for the union S. Inside the { } are the permissible types.

- At any given time however the union can hold one value for ONLY one of its members.

Unions in C/C++

- To declare an instance of the `union`'s tag we use the following syntax;

```
union S MyUnion;
```

  This will create an instance of the `union`.

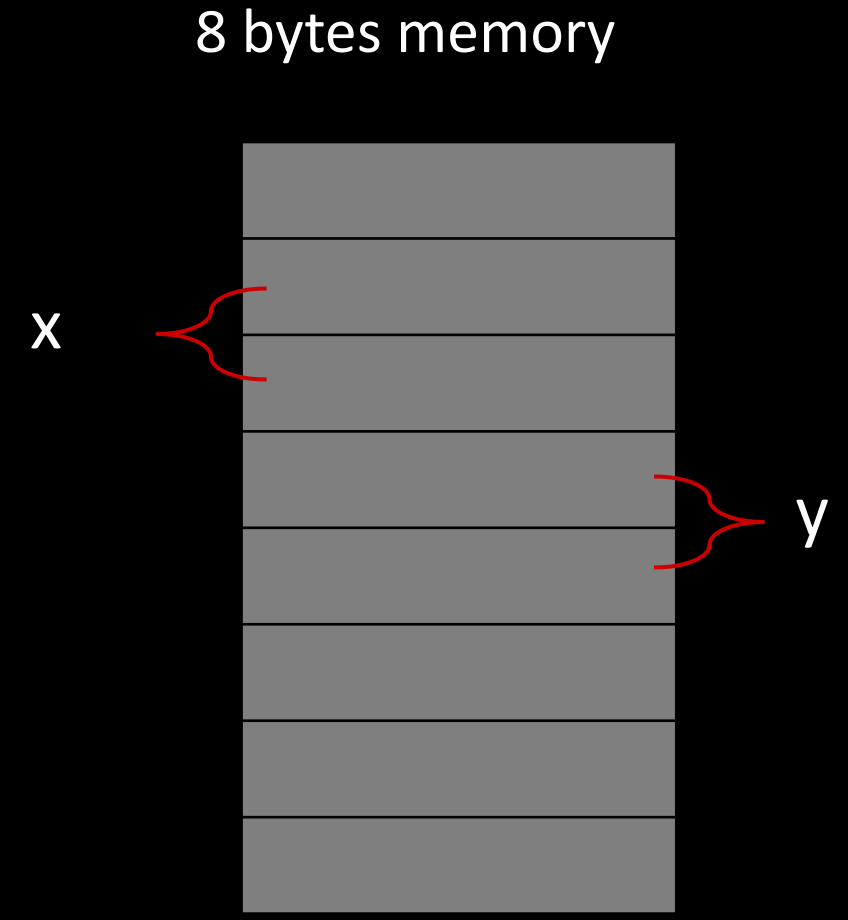
  To access the members of a `union` we can use the same operators as you have seen for `structs`.

Unions

- When you create a instance of a `union`, all members of the `union` share the same memory region.
- That is, the largest of all members is what the `union` consumes in the computers memory.
- Because of this, we can get some funny results.

Unions

```
union S
{
  int x;
  double y;
};
```



Unions in C/C++

```
union Numbers{
  char c;
  int a;
  float d;
};
int main()
{
  union Numbers MyNum;
  MyNum.c = 'A';
  cout << "A char: " << MyNum.c << endl;
  cout << "A int: "  << MyNum.a << endl;
  cout << "A float " << MyNum.d << endl;
  return 0;
}
```

A char is A
A int is 1090519045
A float is 8.000005
A double is 131072.749878

Unions

- In a Union there is no mechanism that automatically records which member has been set. It is up to the programmer to remember which member was set.

```
union Number{
        char c;
        int i;
        long l;
        double d;
};

struct WhichMember{
        int Which;
        union Number MyNumber;
};
```

As you can see we have wrapped the union within a struct. The struct describes an additional member which can be used to keep track of which member in the union is being used.

Unions - another example

```
struct JetType{
  int MaxPassengers;
  ...;
};
struct HeliType{
  int LiftCapacity;
  ...;
};
struct PlaneType{
  double MaxPayload;
  ...;
};
```

```
union AircraftType{
  JetType Jetu;
  HeliType Heliu;
  PlaneType Plane;
};

struct Aircraft{
  int Type;
  int Speed;
  AircraftType Description;
};
```

Unions

- In that example we have defined many different kinds of planes.
- We have formed a union called 'AircraftType' which combines the planes.
- We then use a `struct` to help us identify which plane we have been set in the union.
- The end result here is we have saved a considerable chunk of memory.

Bit Fields

- Within structures and unions it is possible to define the size of a member in bits.
- This enables us to:
  - economize on the storage for a structure's members
  - access the individual bits of storage (same general purpose as bitwise operators)

Bit Fields

- Declare a structure

```
struct bit_example {
   unsigned int field1:4;
   unsigned int field2:8;
   unsigned int field3:4;
}
```

  - A bit field is declared just as any other
  - the data type, but it MUST be an `int`, `signed int` or `unsigned int`
  - The number following the colon (:) is an integer constant gives the width of the bit field
  - The width cannot exceed the size in bits of the integer type.
  - Adjacent bit field members are packed into machine words (although byte order may vary on different systems).

```
/* mixture of regular and bit fields */
struct sample {
  unsigned int num1:16;    /* num1 and num2 are packed */
  unsigned int num2:16;
  int num3;                /* num3 is a regular member */
  char letter;             /* ... */
} var;
var1.num1 = 99;
var.num2 = 11;
var.num3 = -99;
var.letter = 'A';
/* if the system has 32-bit unsigned int,
then num1 and num2 are compacted into one unsigned int) */
```

Bit Fields

- C/C++ gives us greater control over this by allowing us to specify ‘holes’ in sequence and control whether or not the next bit field members go into another machine word.

  If the name of a member is omitted but the width is left then the compiler leaves a hole surrounded by other members.

  Furthermore a member can be of size 0. It requests assignment of previous members to one machine word. Succeeding bit fields will be assigned to a new word.

Bit Fields

- Here is an example:

```
struct bitfield
{
        int int3bits : 3;
        usigned int int1b : 1;
        int int7b : 7;
        int : 2; // make 2 bit hole
        int int2bits : 2;
        int : 0; // put next member in new word
        int int4bits : 4, int5bits : 5;
};
```

Notice how we have a 0 and a 2. The 2 denotes a hole whilst the zero means to put further members in a new word and attempt to put previous ones in a word.

Segmentation Faults and Bus Errors

- Segmentation faults
  - Segmentation faults occur when you try to use memory which does not belong to you, typically
    - out of bounds array references
    - reference through un-initialized or mangled pointers

```
int Ary[5];
cin >> Ary[10];
```

```
char *p;
cout << *p;
```

Segmentation Faults and Bus Errors

- Bus Errors
  - A bus error is slightly different, we get a bus error when we have memory access errors.

    It is associated with manipulating or using unaligned words.

    An example would be a double, a double requires 8 bytes and have an alignment of four bytes (32 bit machine).

    This means their address must be a multiple of 4 (on a word boundary). If a program tries to access such memory using an address that is not a multiple of 4 then we get a bus error.

```
char Array[10];
double *p = &Array[1]; // p points at second char
cout << *p; // bus error when dereferenced
```

Programming in C++

- Procedure Programming
  - function based
- Object-based Programming
  - Object-based programming is a style based on encapsulation of data and functions that work on that data into objects. Object-based programs are usually imperative
- Object-oriented programming
  - A style of programming that supports encapsulation, inheritance, and polymorphism