CSE 165/ENGR 140
Intro to Object Orient Program
Lecture 5 – C in C++
Lab #1
  ◦ Due date: 2/5 (Thursday) at 11:59PM.

Reading assignment
  ◦ Ch. 3
Types: casting

- Types can be converted by C-like type-casts in parenthesis.

```cpp
//: C03:FunctionCallCast.cpp
int main() {
    float a = float(200);
    // This is equivalent to:
    float b = (float)200;
}
```

- In C++ we should use:
  - `static_cast`: simple casts for type conversion etc
  - `const_cast`: to remove const
  - `reinterpret_cast`: to cast an object to something completely different
  - `dynamic_cast`: type-safe cast with run-time checking, can be used only with pointers and references to objects
Types: reinterpret_cast

```cpp
//: C03:reinterpret_cast.cpp
#include <iostream>
using namespace std;
const int sz = 100;

struct X { int a[sz]; };

void print(X* x) {
    for (int i = 0; i < sz; i++)
        cout << x->a[i] << ' ';
    cout << endl << "-------------------" << endl;
}

int main() {
    X x;
    print(&x);
    int* xp = reinterpret_cast<int*>(&x); // Cast to integer
    for (int* i = xp; i < xp + sz; i++)
        *i = 0;

    // Can't use xp as an X* unless you cast it back:
    print(reinterpret_cast<X*>(xp));
}
```
class BaseClass { ... };

class Class1 : public BaseClass {...}; // BaseClass “derives” Class1

class Class2 {...}; // BaseClass does not derive Class2

BaseClass *pb; // pointer to BaseClass
Class1 *p1; // pointer to Class1

p1 = static_cast<Class1*>(pb); // Ok as long as we know pb can point to Class1
p1 = (Class1*)(pb); // C-style also ok, same as static_cast<>

Class2 *p2; // pointer to Class2
p2 = static_cast<Class2*>(pb); // Compiler error, can't convert
p2 = (Class2*)(pb); // No compiler error...
    // Same as reinterpret_cast<>

p2 = reinterpret_cast<Class2*>(pb); // No compiler error.
    // A reinterpret_cast<> can only be justified in rare situations.
    // When you use a reinterpret_cast<> you tell the compiler that you
    // need a non-usual type of casting and that you know what you are doing...
Data types: creation of new types

- We can define our own data types based on existing data types.
  - `typedef existing_type new_typename;`

```c
typedef float coordinate;
typedef char * pointer_char;
typedef double dimension[3];
pointer_char p_char; // instead of char * p_char;
coordinate x;      // instead of float x;
coordinate y;      // instead of float y;
dimension a, b, c, d; // instead of double a[3], b[3], ...
```
Data types: structure

- Structure is a group of elements under one name.
- Elements can be of different data type.
- Data elements are called “members”.
- Structure is defined for the rest of the program.

```c
struct structure_name
{
    member_type_1  member_name_1;
    member_type_2  member_name_2;
    ...
    member_type_n  member_name_n;
} object_names;
```
struct vehicle
{
    string make;
    string model;
    int year;
} car, truck, bike;

struct fruit
{
    double weight;
    float price;
    bool ripe;
};

fruit apple;
fruit banana, pear;
Member access in structure

- We can refer to
  - Whole object: `object_name`
  - Each member: `object_name.member_name`

- Examples
  - `bike` (vehicle)
  - `car.model` (string)
  - `car.year` (int)
  - `apple.weight` (double)
  - `pear` (fruit)
  - `banana.ripe` (bool)

```c
struct vehicle
{
    string make;
    string model;
    int year;
} car, truck, bike;

struct fruit
{
    double weight;
    float price;
    bool ripe;
};
fruit apple;
fruit banana, pear;
```
Structure assignment

- To assign the whole object
  - fruit apple;
  - apple = { 0.22222, 1.75, false };
  - fruit peach = { 2./3., 2.50, true };

- To assign individual members
  - vehicle car;
    - car.make = “Acura”;
    - car.model = “NSX”;
    - car.year = 2015;

```c
struct fruit
{
    double weight;
    float price;
    bool ripe;
};

struct vehicle
{
    string make;
    string model;
    int year;
};
```
```cpp
//: C03:SimpleStruct.cpp
struct Structure1 {
    char c;
    int i;
    float f;
    double d;
};

int main() {
    Structure1 s1; // In C you would need: “struct Structure1 s1;”
    s1.c = 'a';    // Select an element using a '.'
    s1.i = 1;
    s1.f = 3.14;
    s1.d = 0.00093;
}
```
Pointers to structures

- Declaration
  - `structure_name * pointer_name;`
  - `vehicle car;`
  - `vehicle * p_car;`
  - `p_car = &car;`
Pointers to structures

- To access members using a pointer
  - \((*\text{pointer}).\text{member}\)
  - \(\text{pointer->member}\)

```c
(*p_car).model = "NSX";
p_car->model = "NSX";
p_car->make = "Fiat";
(*p_car).model = "500";
fruit * p_apple = &apple;
fruit * p_peach = &peach;
p_apple->weight = p_peach->weight;
(*p_apple).weight = p_peach->weight;
```
Pointers to structures

- \((\ast\text{pointer}).\text{member} \neq \ast\text{pointer}\text{.member})\)

- \(*\text{object.p_member} = *(\text{object.p_member})\)
  - The member of an object is a pointer
  - See operator precedence:
Pointers to structures example

//: C03:SimpleStruct3.cpp
// Using pointers to structs
typedef struct Structure3 {  // skip typedef in C++
    char c;
    int i;
    float f;
    double d;
} Structure3;

int main() {
    Structure3 s1;
    Structure3* sp = &s1;
    sp->c = 'a';
    sp->i = 1;
    sp->f = 3.14;
    sp->d = 0.00093;
}
Data types: enumerations

- Allows us to create new data types with predefined values.

```cpp
enum enum_name
{
    enum_element_1 = enum_value_1,
    enum_element_2 = enum_value_2,
    ...
    enum_element_n = enum_value_n
};

color myColor;
myColor = BLU;
mood myMood = SLEEPY;
```
Data types: enumerations

// C03:Enum.cpp
// Keeping track of shapes

enum ShapeType {
    circle,
    square,
    rectangle
}; // Must end with a semicolon like a struct

int main() {
    ShapeType shape = circle;
    // Activities here....
    // Now do something based on what the shape is:
    switch (shape) {
        case circle: /* circle stuff */ break;
        case square: /* square stuff */ break;
        case rectangle: /* rectangle stuff */ break;
    }
}
Data types: enumerations

- Enumerators offer type checking and must always preferred than defines!

```c
enum ShapeType { circle=10, square=20, rectangle=50 };  

enum snap { crackle=25, pop=10 };  

void draw ( ShapeType t )  
{
    // call drawing commands according to type here
}

void main ()  
{
    snap s = pop;  
    draw ( s );  // will generate warning or error
}
```
Data types: unions

- Unions allow one location of memory to be accessed as different data types.
- The size of a union is the one of its largest member element.
- Modification of one of the members will affect the value of all members.
- It is not possible to store different values for individual members.
//: C03:Union.cpp
#include <iostream>
using namespace std;

union Packed { // Declaration similar to a struct or class
    char i;     // The union will be the size of a
    short j;   // double, since that's the largest element
    int k;
    long l;
    float f;
    double d;
};

int main() {
    cout << "sizeof(Packed) = " << sizeof(Packed) << endl;
    Packed x;
    x.i = 'c';
    cout << x.i << endl;
    x.d = 3.14159;
    cout << x.d << endl;
}
Array: collection of elements of the same type.

Array size: how many elements—integer number n.

```c
int a[10]; // a[0], .... a[9]

int n=10;
int A[n]; // A[0], ..... A[n-1]
```
Array—Types and initialization

```c
char c[4];
short s[5];
int y[2];
long l[7];
float f[10000];
double x[n];
bool b[2];

char c[] = { 'a', 'b', 'c', 'd' };
int n = 10;
int i = 3;
int a[5] = { 4, 3, 2, 1, 0 };
double x[n];
x[0] = 2.1;
x[i] = .1e5;
x[4] = i / 2;
x[a[i]] = a[i];
x[i] = i;
bool b[2] = {false, true};
```
# Data types: arrays

//: C03:Arrays.cpp
#include <iostream>
using namespace std;

int main() {
    int a[10];
    for(int i = 0; i < 10; i++) {
        a[i] = i * 10;
        cout << "a[" << i << "] = " << a[i] << endl;
    }
}

//: C03:StructArray.cpp
typedef struct ThreeDPoint{
    int i, j, k;
};

int main() {
    ThreeDPoint p[10];
    for(int i = 0; i < 10; i++) {
        p[i].i = i + 1;
        p[i].j = i + 2;
        p[i].k = i + 3;
    }
}
Pointers and arrays

- Identifier of an array equivalent to the address of array’s first element.
- The i-th element of \( a \) can be dereferenced either way:
  - \( a[i] \)
  - \( *(a+i) \)
- Example:
  - \texttt{int a[20];} // \( a \) is a pointer to \( a[0] \)
  - \texttt{a[5] = 0;} // \( a[\text{offset of 5}] = 0 \)
  - \texttt{*(a+5) = 0;} // pointed by \( (a+5) = 0 \)
Examples:
- int numbers[20];
- int * p;
- p = numbers;

- double d[] = {0., 1., 2.} ;
- double* p_var = d;       //points to array d
- double* p_array[3];     //array of 3 pointers
- p_array[0] = &d[0];      //1st pointer is the address of d[0]
// What is the difference between *ip++, *(ip++) and +++ip?
// What would ip+=4 do?
Preprocessor debugging flags

- Use `#define DEBUG` if we want to debug.
- We can set up a if-else condition to turn on/off our debugger.

```c
#define DEBUG // Probably in a header file
#ifdef DEBUG // Check to see if flag is defined
/* debugging code here */
#endif // DEBUG
```
```cpp
// Ex2: Using macros for printing
// (C03: StringizingExpressions.cpp)
#define DEBUG // Keep this line if we want to debug
#include <iostream>
using namespace std;

#if defined(DEBUG)
#define P(A) cout << #A << " : " << (A) << endl;
#else
#define P(A) #endif

int main() {
    int a = 1, b = 2, c = 3;
    P(a); P(b); P(c);
    P(a + b);
}
```

Output:

```
a: 1
b: 2
c: 3
a + b: 3
```
C assert() macro

- When you use `assert()`, you give it an argument that is an expression you are “asserting to be true.”
- The preprocessor will test the assertion.
- The program will stop after issuing an error message.

```cpp
//: C03:Assert.cpp
// Use of the assert() debugging macro
#define NDEBUG // Remove this line if you want to debug
#include <cassert> // Contains the macro
using namespace std;

int main() {
    int i = 100;
    assert(i != 100); // Fails
}
```

Output:
Assertion failed: i != 100, file C:\Users\Daniel Leung\Desktop\lect5\lect5.cpp, line 9

This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information.