CPP Programming: Session 3

Anupama Chandrasekhar (Aug 16 2018)

Pointers

- Pointers are special types of variables that hold an address.

```
int a = 100;
int *aptr = &a; // wrt pointers, & is the address-of operator

*aptr = 25; // wrt pointers, * is the dereference operator
```

- Why do you need pointers?
  - To modify outside objects within a function
  - Other stuff??

`void*`

- Pointer to an object of unknown type
- To use **`void*`**, we need to explicitly convert it to another type

```
void f(int* pi)
{
    void* pv = pi;   // ok: implicit conversion of int* to void*
    *pv;             // error: can't dereference void*
    ++pv;            // error: can't increment void* (the size of the object pointed to is unknown)

    int* pi2 = static_cast<int* >(pv);          // explicit conversion back to int*
    double* pd1 = pv;                           // error
    double* pd2 = pi;                           // error
    double* pd3 = static_cast<double*>(pv);     // unsafe
}
```

C++11

- The literal **`nullptr`** doesn’t point to any object.

Arrays

- For a type **T**, **T[size]** is the type "array of **size** elements of type **T**." The elements are indexed from **0** to **size–1**.

```
float v[3];       // an array of three floats: v[0], v[1], v[2]
char* a[32];      // an array of 32 pointers to char: a[0] .. a[31]
```

- There is no array assignment and arrays implicitly convert to the pointer of its first element in many cases.
- One of the most widely used kinds of arrays is a zero-terminated array of **char**. That's the way C stores strings, so a zero-terminated array of **char** is often called a *C-style string*

```
void foo()
{

    int a2 [20];            //20 ints on the stack
    int*p = new int[40];    //40 ints on the free store
     //...

    delete[] p;
}
```

Array Initializers

- Array initializing:

```
int v1[] = { 1, 2, 3, 4 };
char v2[] = { 'a', 'b', 'c', 0 };
char v3[2] = { 'a', 'b', 0 };      // error: too many initializers
char v4[3] = { 'a', 'b', 0 };     // OK
int v5[8] = { 1, 2, 3, 4 };     // OK, 0 for rest
int v6[8] = v5;   // error: can't copy an array (cannot assign an int* to an array)
v6 = v5;          // error: no array assignment
```

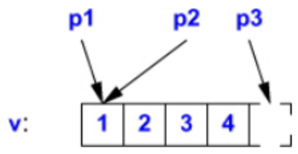
- You cannot pass arrays by value
- Arrays of characters can be conveniently initialized using a string literal.

```
const char name[5] = "Sara";
const char* name2 = "Sara";
```

Pointers and Arrays

- C++ pointers and arrays are closely related

```
int v[] = { 1, 2, 3, 4 };
int* p1 = v;            // pointer to initial element (implicit conversion)
int* p2 = &v[0];        // pointer to initial element
int* p3 = v+4;          // pointer to one-beyond-last element
```




```
void fp(char v[], int size)
{
    for (int i=0; i!=size; ++i)
        use(v[i]);              // hope that v has at least size elements

    for (int x : v)
        use(x);                 // error: range-for does not work for pointers

    const int N = 7;
    char v2[N];

    for (int i=0; i!=N; ++i)
        use(v2[i]);

    for (int x : v2)
        use(x);                 // range-for works for arrays of known size
}
```

Pointers and Arrays

- Arrays cannot be passed by value

```
void comp(double arg[10])          // arg is a double*
{
    for (int i=0; i!=10; ++i)
        arg[i]+=99;
}

void f()
{
    double a1[10];
    double a2[5];
    double a3[100];

    comp(a1);
    comp(a2);       // disaster!
    comp(a3);       // uses only the first 10 elements
};

void comp(double* arg)
{
    for (int i=0; i!=10; ++i)
        arg[i]+=99;
}
```

Pointer and `const`

- To declare the pointer itself const use `*const`

```
const int model = 90;                   // model is a const
const int v[] = { 1, 2, 3, 4 };         // v[i] is a const
const int x;                            // error: no initializer
void f()
{
    model = 200;        // error
    v[2] = 3;           // error
}

void g(const X* p)
{
     // can't modify *p here
}

void h()
{
    X val;          // val can be modified here
    g(&val);
     //...
}
```

References

- References fix some problems that pointers have
  - Using ***ptr_obj** instead of **obj** and **ptr_obj->m** instead of **obj.m**
  - Pointers can point to different objects at different times
  - We need to be careful with pointers as they can point objects other than ones we are expecting or to **nullptr**
- References are syntactically cleaner
  - Access a reference with exactly the same syntax as the name of the object
  - Reference always refers to the object it was initialized
  - There is no “null reference”. Think of a reference as a alternative name of an object.

References

- To reflect lvalue/rvalue and const/non-const differences, there are three kinds of references:
  - Lvalue references: refer to objects whose values we want to change
  - Const references: refer to objects we only want to read from and not modify
  - Rvalue references: refer to objects whose value we do not need to preserve after we have used it (e.g., temporaries)

- An rvalue reference refers to a temporary object, which the user of the reference can (and typically will) modify, assuming that the object will never be used again. We want to know if a reference refers to a temporary, because if it does, we can sometimes turn an expensive copy operation into a cheap move operation

References

- && means rvalue reference and it is used to implement destructive read

```
string var {"Cambridge"};
string f();

string& r1 {var};            // lvalue reference, bind r1 to var (an lvalue)
string& r2 {f()};            // lvalue reference, error: f() is an rvalue
string& r3 {"Princeton"};    // lvalue reference, error: cannot bind to temporary
string&& rr1 {f()};          // rvalue reference, fine: bind rr1 to rvalue (a temporary)
string&& rr2 {var};          // rvalue reference, error: var is an lvalue
string&& rr3 {"Oxford"};     // rr3 refers to a temporary holding "Oxford"
const string& cr1 {"Harvard"}; // OK: make temporary and bind to cr1
```

Structures, Unions and Enumerations

- The key to good object oriented programming is using proper user defined types.
  - A **struct** is a sequence of elements or arbitrary types.
  - A **union** is a struct that holds the value of just one of its elements at any one time.
  - An **enum** is a type with a set of name constants

  - An **enum class** is an enum where the enumerators are within the scope of the enumeration and no implicit conversions to other types is provided.

Structures

- An array is an aggregate of elements of the same type, in its simplest form a **`struct`** is an aggregate of elements of arbitrary types.

```
struct Address {
        const char* name;           //"Jim Dandy"
        int number;                 //61
        const char* street;         //"South St"
        const char* town;           //"New Providence"
        char state[2];              //'N' 'J'
        const char* zip;            //"07974"
};
```

- Variables of a structure are declared just like the built in types and individual elements can be accessed thus:

```
void f()
{
        Address jd;
        jd.name = "Jim Dandy";
        jd.number = 61;
}
```

Structures

- Variables in a structure can be initialized thus:

```
Address jd = {"Jim Dandy", 61, "South St", "New Providence",{'N','J'}, "07974"};
```

- Structures are often accessed through pointers using p->m == (*p).m

```
void print_addr(const Address* p)
{
    cout << p->name << '\n'
        << p->number << ' ' << p->street << '\n'
        << p->town << '\n'
        << p->state[0] << p->state[1] << ' ' << p->zip << '\n';
}
```

- Objects of structure types can be assigned, passed as arguments and returned from a function. Other operators like == and != are not defined by default but can be defined by a user.

`struct` Layout

- An object of a struct holds its members in the order in which they were declared.

```
struct Readout {
	char hour;      //[0:23]
	int value;
	char seq;
};
```

- The size of the **struct** object may not be the sum of the size of its members due to the alignment requirements of the machine. You can find the size using the sizeof operator.
- While you can change the order of declaration to optimize storage, better to optimize of readability unless there is a need to optimize on storage.

`struct` and `class`

- A **`struct`** is a class where the members are public by default. So a **`struct`** can have member functions, constructors etc.
- The name of a struct is immediately available after it has been encountered. However to declare an object the full definition is needed.

```
struct Node
{
    Node* left;
    Node* right;
    int data;
};
```

- So name is sufficient as long as member name or size is not needed.

Structures and Arrays

- **`struct`** with array and array of **`struct`**s

```
struct Point {
    int x,y;
};

Point points[3] {{1,2},{3,4},{5,6}};
int x2 = points[2].x;

struct Array {
    Point elem[3];
};

Array points2 {{1,2},{3,4},{5,6}};
int y2 = points2.elem[2].y;
```

- STL provide a fixed size container **`std::array`** (covered in a later session). Proper object type that doesn't implicitly convert to pointer to the first element

`struct` : Type Equivalence and bit fields

- Two structures are different types even if they have the same members

```
struct S1 { int a; };

struct S2 { int a; };

S1 x;

S2 y = x;  // error: type mismatch
```

- Fields:

```
struct PPN {        // R6000 Physical Page Number
    unsigned int PFN : 22;   // Page Frame Number
    int : 3;                 // unused
    unsigned int CCA : 3;    // Cache Coherency Algorithm
    bool nonreachable : 1;
    bool dirty : 1;
    bool valid : 1;
    bool global : 1;
};
```

union

- A union is a struct in which all members are allocated at the same address so that the union occupies only as much space as its larges member

```
enum Type { str, num };

struct Entry {
	char* name;
	Type t;
	char* s;   // use s if t==str
	int i;     // use i if t==num
};

void f(Entry* p)
{
	if (p->t == str)
		cout << p->s;
	//...
}

union Value {
	char* s;
	int i;
};
```

Enum and Enum Classes

- There are two kinds of enumerations:
  - C++11 Enum classes, for which the enumerator names are local to the enum and their values do not implicitly convert to other types
  - “Plain enums” for which the enumerator names are in the same scope as the enum and their values implicity convert to integers
  - Pro Tip In general prefer enum classes for fewer surprise behaviours

Enum Classes

C++11

- Enum class is a scoped and a strongly typed enum

```
enum class Traffic_light { red, yellow, green };
enum class Warning { green, yellow, orange, red }; // fire alert levels

Warning a1 = 7;                  // error: no int->Warning conversion
int a2 = green;                  // error: green not in scope
int a3 = Warning::green;         // error: no Warning->int conversion
Warning a4 = Warning::green;     // OK

void f(Traffic_light x)
{
     if (x) { / ... /}                          // error : Traffic_light is not a bool
     if (x == 9) { /* ... */ }                  // error: 9 is not a Traffic_light
     if (x == red) { /* ... */ }                // error: no red in scope
     if (x == Warning::red) { /* ... */ }       // error: x is not a Warning
     if (x == Traffic_light::red) { /* ... */ } // OK
}
```

Plain enums

```
enum Traffic_light { red, yellow, green };
enum Warning { green, yellow, orange, red };  // fire alert levels
 // error: two definitions of yellow (to the same value)
 // error: two definitions of red (to different values)

Warning a1 = 7;                   // error: no int->Warning conversion
int a2 = green;                   // OK: green is in scope and converts to int
int a3 = Warning::green;          // OK: Warning->int conversion
Warning a4 = Warning::green;      // OK

void f(Traffic_light x)

{
    if (x == 9) { /* ... */ }                   // OK (but Traffic_light doesn't have a 9)
    if (x == red) { /* ... */ }                 // error: two reds in scope
    if (x == Warning::red) { /* ... */ }        // OK (Ouch!)
    if (x == Traffic_light::red) { /* ... */ }  // OK
}
```

Statements

- Statements specify an order of execution.
- Declarations, assignments, selection statements, iteration statements etc.
- Declaration (repeat): Declare just before use and preferably initialize.

```
void foo()
{
    std::string s1;
    //...
    s1 = "Fee Fie Foe Fum";

    string s2 {"I smell the blood of an Englishman"};
}
```

Statements : Selection Statements

- **`if`** statements : Evaluated if the condition evaluates to true, the condition might be implicitly converted to bool in case of integers, pointers etc.

```
if (x) // for integer, equivalent if (x!=0)

if (p) // for pointers, equivalent if (p!=nullptr)
```

- Logical operators **`&&  ||  !`** are commonly used in conditionals, **`&&`** and **`||`** will not evaluate their second argument unless needed.
- A more direct way to express the intent of choosing between two alternatives is **`(condition)? result1 : result2;`**

Statements: Selection Statements

```cpp
int max(int a, int b)
{
    return (a>b) ? a : b;
}

int foo(int a)
{
    if (a)
    {
        int x = a++;
    }
    else
    {
        x = a--; // x not in scope
    }

    return x; //x not in scope
}
```

```cpp
int& max(int& x1, int& x2, int& x3)
{

    return (x1>x2)? ((x1>x3)?x1:x3) : ((x2>x3)?x2:x3);

}
```

Statements: **`switch`** Statements

```
switch (val) {            // beware

case 1:

      cout << "case 1\n";

case 2:

      cout << "case 2\n";

default:

      cout << "default: case not found\n";

}
```

```
switch (action) {// handle (action,value) pair

case do_and_print:
      act(value);
       // no break: fall through to print
case print:
      print(value);
      break;
 //...
}
```

- **`break`** is a common way to terminate a **`case`**, **`return`** too in some cases
- Exercise: What is the compiler error in switch.cpp mean?

Statements : Iteration

- **`while`** (*condition*) *statement*
  `do` *statement* **`while`** (*expression*) ;
  `for` (*init; termination;expression*) *statement*
  C++11 **`for`** (*for-init-declaration* : *expression*) *statement*

```
for (;;) {  //"forever"
        //...
}

while(true) {      //"forever"

        //...
}
```

- Use **`break`** to leave the loop body in the middle. **`continue`** is used to skip the rest of the loop body

Statements: Comments and Indentation

- There are two forms of comments:
  - *line comments:* **//** the comment extends to the end of the line
  - *block comments:* **/*** the comment extends to the end-of-comment marker ***/**
  - Block comments do not nest

Functions

- Function Declaration : **`<Return type> FunctionName (args)`**

```
Elem* next_elem();          // no argument; return an Elem*
void exit(int);             // int argument; return nothing
double sqrt(double);        // double argument; return a double
```

```
double s2 = sqrt(2);            // call sqrt() with the argument double{2}
double s3 = sqrt("three");      // error: sqrt() requires an argument of type double
```

- Purpose of a function is to break up complicated computations into easily digestable chunks. Avoid very long functions.
- Like use a dot product to perform matrix multiplication versus using nested loops.

Anatomy of a function declaration

- Name of the function
- Argument list which might be empty **`void PrintWelcomeMessage()`**
- Return type, could be void, could be **`auto`** from C++14

C++11

- **`inline,`** function calls should be implemented by inlining the function body
- **`Constexpr`**, can be evaluated at compile time if inputs are constant expressions
- **`Noexcept,`** the function may not throw an exception
- Linkage specification like **`static`**
- **`[[no return]]`**, function will not return using normal call/ret mechanism

Function Definitions

- A function that is called must be defined somewhere
- Naming arguments in declarations that are not definitions is optional
- Pro Tip We can indicate that an argument is not used in a function definition by not naming it. Use: Future proofing function signature
- What else can we call? Constructors, Destructors, Function Objects, Lambda expressions
- Returning values: Prefix and Suffix return types

```
string to_string(int a);            // prefix return type

auto to_string(int a) -> string;    // suffix return type
```

Function Returns

```
int f3() { return 1; }         // OK
void f4() { return 1; }        // error: return value in void function

int f5() { return; }           // error: return value missing
void f6() { return; }          // OK

int fac(int n)
{
    return (n>1) ? n*fac(n-1) : 1;
}
```

- Function that calls itself is recursive.
- There can be more than one return statement in a function.
- Exercise : Modify the **`int fac(int)`** function to use multiple returns and make sure it works.

Understanding Function calls

- Every time a function is called a new copy of its arguments and local variables are created. The store is reused after the function returns. A pointer to a non-static local variable should not be returned as its contents will vary unpredictably (most compilers should warn about this)

```
int* fp()
{
    int local = 1;
    // ...
    return &local;   // bad
}
```

- A void function can return a void function.
- There are 5 ways to exit a function: 1. Executing a return 2. Falling off the end 3. Throwing an exception, not caught locally 4. Throwing an except that was not caught locally in a **noexcept** function 5. Directly invoking a function call that doesn't return

Pro Tip

- A function that doesn't return "normally" can me marked **[[noreturn]]**

`inline` Functions

- The inline specifier is a hint to the compiler that it should attempt to generate the code for the function call than laying down the code for the function once and then calling it through the usual function call mechanism.
- The complexity of inlining means that the compiler cannot be guarantees that the function will actually be inlined
- Inlining does not change the semantics of a function, the function and its static variables still have unique address
- If an **`inline`** function has more than one definition, all the definitions should be identical.

C++11

`constexpr` Functions

- In general functions cannot be evaluated at compile time, by specifying a function as constexpr, we indicate that we want to be able to use it in constant expressions if give constant expressions as arguments.
- For constexpr, the functions should:
  - Consist of a single return statement
  - No loops or local variables, can refer to non local objects but cannot modify them
  - No sideeffects

`constexpr` Functions

- Spot the errors

```
int glob;
constexpr void bad1(int a)      // error: constexpr function cannot be void
{
    glob = a;               // error: side effect in constexpr function
}

constexpr int bad2(int a)
{
    if (a>=0) return a; else return -a;     // error: if-statement in constexpr function
}

constexpr int bad3(int a)
{
    int sum = 0;                                    // error: local variable in constexpr function
    for (int i=0; i<a; ++i) sum +=fac(i); return sum;    // error: loop in constexpr function
    return sum;
}
```

`constexpr` contd.

- Determining if an expression is **`constexpr`** can be tricky

```
constexpr const int* addr(const int& r) { return &r; }    // OK

static const int x = 5;
constexpr const int* p1 = addr(x);        // OK
constexpr int xx = *p1;                   // OK

static int y;
constexpr const int* p2 = addr(y);        // OK
constexpr int yy = *p2;                   // error: attempt to read a variable

constexpr const int* tp = addr(5);        // error: address of temporary
```

[[no return]]

- [[..]] is called an attribute and specifies some property of the entity that precedes it.
- Placing a **`[[no return]]`** in front of a function implies that it will not return, if despite that a function returns the behavior is undefined.

```
[[noreturn]] void exit(int);      // exit will never return
```

Functions : Local Variable

- A local variable is initialized when the thread of executions reaches its definition.
- If the variable is not **`static`** each invocation has its own copy.
- If the variable is **`static`** then a single statically allocated object will be used to represent it in all calls of the function. It is initialized only the first time the thread of execution reaches its definition.
- It is a way to preserve a value between calls

Functions : Argument Passing

- Unless the formal argument is a reference, a copy of the actual argument is passed to the function.
- What about passing pointers? See Argpassing_ptr.cpp
- Passing by reference, doesn't make a copy ..provides an alias(reference) to the original object
- Passing by reference while efficient might make the program error prone if you don't expect to modify the original object. Protect with **`const`** wherever possible to denote read only.

Pro Tip

```
void foo (Large& arg)
{
    // assume that the value of arg will be changed
}

void bar (const Large& arg)
{
    // value or arg cannot be changed except
    // by explicit type conversion
}
```

Argument Passing: Rules of Thumb

- [1] Use pass-by-value for small objects.
- [2] Use pass-by-**const**-reference to pass large values that you don’t need to modify.
- [3] Return a result as a **return** value rather than modifying an object through an argument.
- [4] Use rvalue references to implement move and forwarding.
- [5] Pass a pointer if “no object” is a valid alternative (and represent “no object” by **nullptr**).
- [6] Use pass-by-reference only if you have to.

Array Arguments

- When an array is passed as an array argument, a pointer to its initial element is passed. => array is not passed by value

```
void odd(int* p);

void odd(int a[]);

void odd(int buf[1020]);
```

- Size of the array is not available to the called function. Preferable to use containers like vector, array or map.

Functions: Overloading and Return Type

- When functions perform the same task on different objects it is convenient to give them the same name. This is overloading.

```
void print(int);           // print an int
void print(const char*);   // print a C-style string
```

- Automatic overload resolution: Resolution happens by comparing types of inputs and give a compile time error is no appropriate overload is found or the resolution is ambiguous.
- Return type is not considered in overload resolution.

Functions: Macros

- First rule about macros: Don't use them unless you have to. Recommended only for conditional compilation.

```
#define PI 3.142

#define SQUARE(a) a*a

void f(int xx)
{
    int y = SQUARE(xx+2);     // y=xx+2*xx+2; that is, y=xx+(2*xx)+2
}
```