Ve 280

Programming and Elementary Data Structures

Developing Programs on Linux;

Review of C++ Basics

Outline

- Developing programs on Linux

- Review of C++ basics

Review

- Compile a program on Linux

g++ -o program source.cpp

= g++ -c source.cpp
  g++ -o program source.o

- Header guard

```
// add.h
#ifndef ADD_H
#define ADD_H
int add(int a, int b);
#endif
```

Header guard to prevent multiple definitions!

Compiling Multiple Source Files

- To compile multiple source files, use command
  - g++ -Wall -o program src1.cpp src2.cpp src3.cpp

Program name

All .cpp files

  - E.g., g++ -Wall -o run_add run_add.cpp add.cpp

- Note: you don't put ".h" in the compiling command
  - I.e., you don't have
    g++ -Wall -o program src1.cpp src1.h src2.cpp src3.cpp
  - Why? ".h" files are already included.
    E.g., run_add.cpp includes add.h

Another Way

- Generate the object codes (.o files) **first**
- Example: g++ -Wall -o run_add run_add.cpp add.cpp
  - **Equivalent** way:

    g++ -Wall -c run_add.cpp # will produce run_add.o
    g++ -Wall -c add.cpp   # will produce add.o
    g++ -Wall -o run_add run_add.o add.o
  - Advantage?
  - Disadvantage?

A Better Way: Makefile

```
all: run_add

run_add: run_add.o add.o
	g++ -o run_add run_add.o add.o

run_add.o: run_add.cpp
	g++ -c run_add.cpp

add.o: add.cpp
	g++ -c add.cpp

clean:
	rm -f run_add *.o
```

- The file name is "Makefile"
- Type "make" on command-line

A Rule

Target: Dependency
<Tab> Command

Don't forget the Tab!

Dependency: A list of files that the target depends on

A Better Way: Makefile

```
all: run_add

run_add: run_add.o add.o
	g++ -o run_add run_add

run_add.o: run_add.cpp
	g++ -c run_add.cpp

add.o: add.cpp
	g++ -c add.cpp

clean:
	rm -f run_add *.o
```

There is a target called "all"

- It is the **default** target
- Its dependency is program name
- It has no command

A Rule

Target: Dependency
<Tab> Command

Usually, there is a target called "clean"

- A **dummy target**. Type "make clean"
- It has no dependency!
- Question: what does "clean" do?

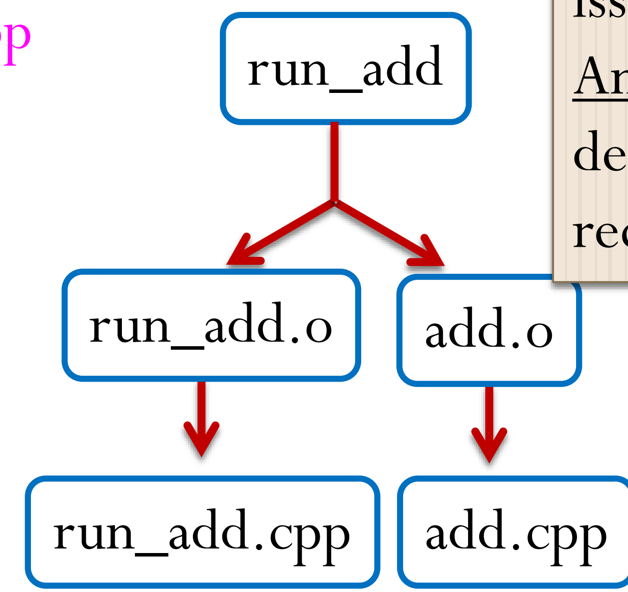
A Better Way: Makefile

```
all: run_add

run_add: run_add.o add.o
	g++ -o run_add run_add.o add.o

run_add.o: run_add.cpp
	g++ -c run_add.cpp

add.o: add.cpp
	g++ -c add.cpp

clean:
	rm -f run_add *.o
```

A Rule

Target: Dependency
<Tab> Command

Dependency Graph



When is a command issued?

<u>Answer</u>: When dependency is more recent than target

Outline

- Developing programs on Linux

- Review of C++ basics

Very Basic Concepts

- Variables
- Built-in data types, e.g., `int`, `double`, etc.
- Input and output, e.g., `cin`, `cout`.
- Operators
  - Arithmetic: +, -, *, etc.
  - Comparison: <, >, ==, etc.
  - x++ versus ++x
- Flow of controls
  - Branch: if/else, switch/case
  - Loop: while, for, etc.

An Example

```
#include <iostream>
using namespace std;
int main() {
  // Calculating the area of a square
  int length, area;
  cin >> length;
  if(length > 0) {
    area = length * length;
    cout << "area is " << area << endl;
  }
  else
    cout << "negative length!" << endl;
  return 0;
}
```

lvalue and rvalue

- Two kinds of expressions in C++
  - **lvalue**: An expression which may appear as either the left-hand or right-hand side of an assignment
  - **rvalue**: An expression which may appear on the right- but not left-hand side of an assignment

- Which of the followings are lvalues? Which are rvalues?
  - `a    // a is an int variable`
  - `10`
  - `a+1 // a is an int variable`
  - `a+b // a and b are two int variables`

Function Declarations vs. Definitions

- Function declaration (or function prototype)
  - Shows how the function is called.
  - Must appear in the code before the function can be called.
  - Syntax:

```
Return_Type Function_Name(Parameter_List);
//Comment describing what function does
```

```
int add(int a, int b); //Comment
```

- Function definition
  - Describes how the function does its task.
  - Can appear before or after the function is called.
  - Syntax:

```
Return_Type  Function_Name(Parameter_List)
{
  //function code
}
```

```
int add(int a, int b) {
      return (a + b);
}
```

Function Declaration

- Tells:
  - return type
  - how many arguments are needed
  - types of the arguments
  - name of the function
  - formal parameter names

Type Signature

Formal Parameter Names

- Example:
  double total_cost(int number, double price);
  // Compute total cost including 5% sales tax on
  // number items at cost of price each

Function Definition

- Provides the same information as the declaration
- Describes how the function does its task

- Example:

function header

```
double total_cost(int number, double price)
{
    double TAX_RATE = 0.05; //5% tax
    double subtotal;
    subtotal = price * number;
    return (subtotal + subtotal * TAX_RATE);
}
```

function body

Function Call Mechanisms

- Two mechanisms:
  - Call-by-Value
  - Call-by-Reference

```
void f(int x)
{
   x *= 2;
}
```

```
void f(int& x)
{
   x *= 2;
}
```

```
int main()
{
   …
   int a=4;
   f(a);
   …
}
```

What will a be?

Array

- An array is a fixed-sized, indexed data type that stores a collection of items, all of the same type.
- Declaration: `int b[4];`
- Accessing array elements using index: `b[i]`
- C++ arrays can be passed as arguments to a function.

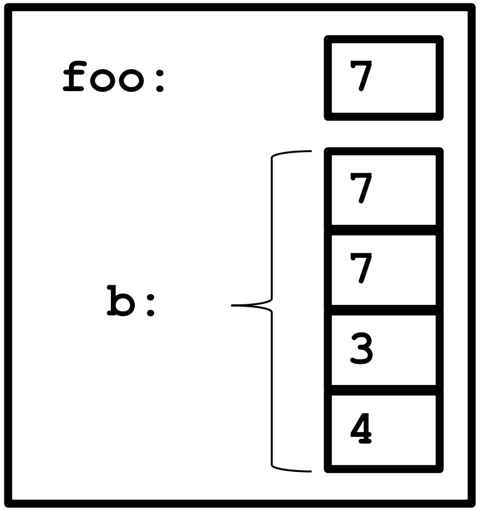
```
int sum(int a[], unsigned int size);
  // Returns the sum of the first
  // size elements of array a[]
```

Array is passed by **reference**.

Array as Function Argument

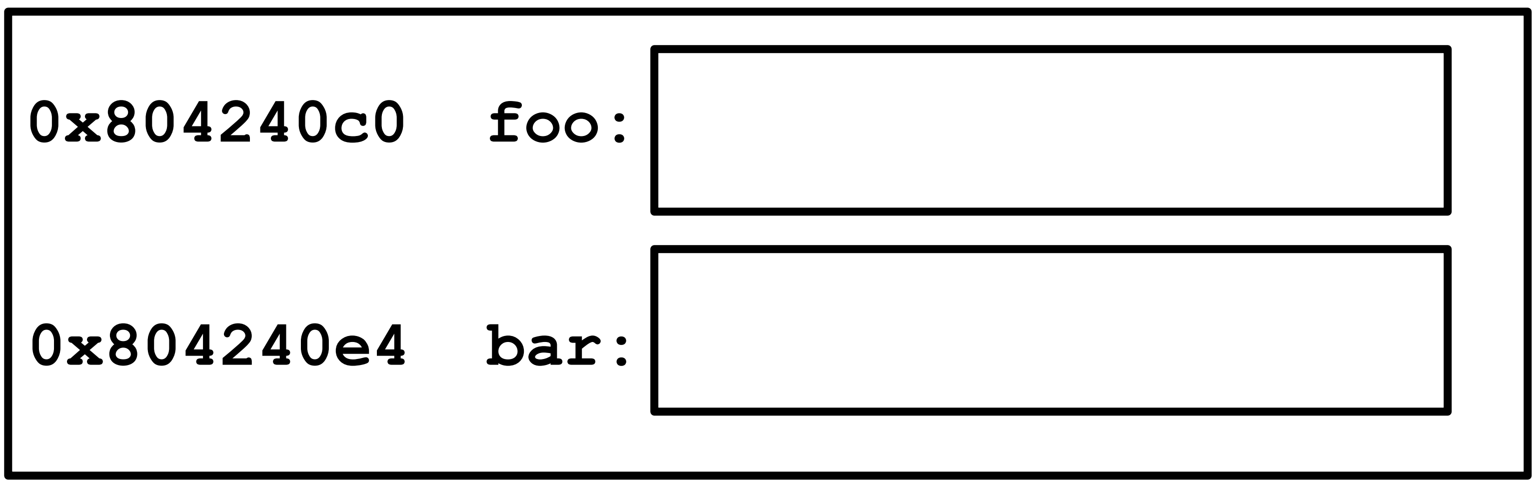
- Using the values below, what would the contents of `b` be after calling `add_one(b, 4)`?

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



Pointers: Working with Addresses

```
int foo = 1;
int *bar;    // Define a pointer
bar = &foo; // addressing operation
*bar = 2;    // dereference operation
```



References

- **Reference** is an **alternative** name for an object.

```
int iVal = 1024;
int &refVal = iVal;
```

- refVal is a reference to iVal. We can change iVal through refVal.

- Reference **must be initialized** using a **variable** of the same type.

```
int &refVal2; // Error: not initialized
int &refVal3 = 10; // Error: 10 is not
                   // a variable
```

References

- There is **no way to rebind** a reference to a different object after initialization.

```
int iVal = 1024;
int &refVal = iVal;
int iVal2 = 10;
refVal = iVal2;
```

- refVal still binds to iVal, not iVal2.

Pointers Versus References

- Both pointers and references allow you to pass objects by reference.

- Any differences between pointers and references?
  - Pointers require some extra syntax at calling time (&), in the argument list (*), and with each use (*); references only require extra syntax in the argument list (&).
  - You can change the object to which a pointer points, but you cannot change the object to which a reference refers.
    - In this sense, pointer is **more flexible**

References Versus Pointers

Example

```
int x = 0;
int &r = x;
int y = 1;
r = y;
r = 2;
```

```
int x = 0;
int *p = &x;
int y = 1;
p = &y;
*p = 2;
```

What's the final values of `x`, `y`, and `r`?

x = 2, y = 1, r = 2

What's the final values of `x`, `y`, and `*p`?

x = 0, y = 2, *p = 2

Pointers

Why use them?

- You might wonder why you'd ever want to use pointers, since they require extra typing, and is error-prone.

- There are (at least) two reasons to use pointers:

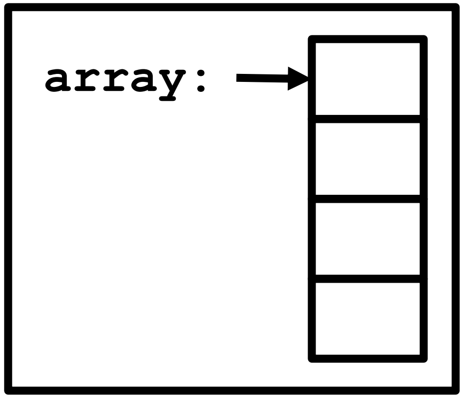
    1. They provide a convenient mechanism to work with arrays.

    2. They allow us to create structures (unlike arrays) whose size is not known in advance.

Pointers and Arrays

- If you look at the **value** of the variable `array` (not `array[0]`) you'll find that it was exactly the same as the **address** of `array[0]`.
- In other words,

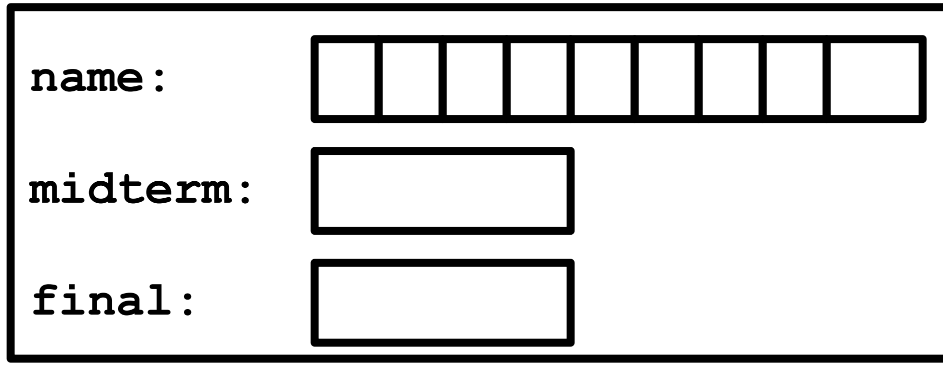
```
array == &array[0]
```



Structs

- Declare a `struct` type that holds grades.

```
struct Grades {
   char name[9];
   int  midterm;
   int  final;
};
```

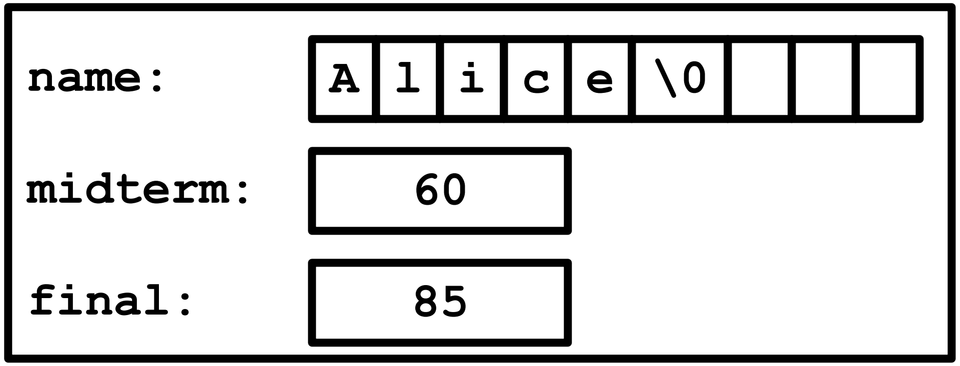


- This statement declares the **type** "struct grades", but does not declare any **objects** of that type.
- We can define single objects of this type as follows:

```
struct Grades Alice;
```

Structs

```
struct Grades {
   char name[9];
   int  midterm;
   int  final;
};
```

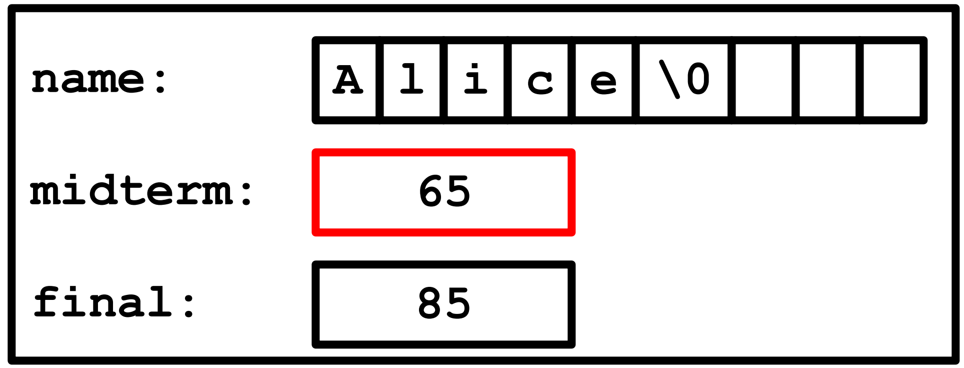


- We can initialize them in the following way:

```
struct Grades Alice= {"Alice", 60, 85};
```

Structs

```
struct Grades {
  char name[9];
  int  midterm;
  int  final;
};
```



- Once we have a struct, we can access its individual components using the “dot” operator:

```
Alice.midterm = 65;
```

- This changes the `midterm` element of `Alice` to 65.

Enum

- Used to categorize data
- Define an enumeration type

```
enum Direction_t {EAST, SOUTH,
                  WEST, NORTH};
```

- To define variables of this type:

```
enum Direction_t dir;
```

- Initialization:

```
enum Direction_t dir = EAST;
```

Enum

If you write

```
enum Direction_t {EAST, SOUTH,
                  WEST, NORTH};
```

then numerically

```
EAST = 0,SOUTH = 1,WEST = 2,NORTH = 3
```

- Using this fact, it will sometimes make life easier

```
enum Direction_t d = EAST;
const string dirname[] = {"east",
    "south", "west", "north"};
cout << "Direction d is "
     << dirname[d];
```

const Qualifier

- Often, a numerical value in a program could have some valid meaning.

```
char name[256];
```

The max size of name string

- Also, that value with the same meaning may appear many times in the program

```
for(i=0; i < 256; i++) ...
```

- If we only use 256, it has two drawbacks
  - The readability is bad.
  - If we need to update max size of a name string from 256 to 512, we need to examine each 256 (some may have other meanings) and update the corresponding ones.
    - It takes time and is error-prone!

const Qualifier

- Instead of just using 256, define a constant, and use the constant:

```
const int MAXSIZE = 256;
char name[MAXSIZE];
```

- Usually, constant is defined as a global variable.
- Property
  - Cannot be modified later on
  - Must be initialized when it is defined

```
const int a = 10;
a = 11; // Error
```

```
const int i;
// Error
```

const Reference

```
const int iVal = 10;
const int &rVal = iVal;
```

- Furthermore, const reference can be initialized to an rvalue

```
const int &ref = 10; // OK
const int &ref = iVal+10; // OK
```

- In contrast, nonconst reference cannot be initialized to an rvalue

```
int &ref = 10; // ERROR
int &ref = iVal+10; // ERROR
```

const Reference

- One popular use of const reference: pass struct/class as the function argument

```
int avg_exam(const struct Grades & gr){
  return (gr.midterm+gr.final)/2;
}
```

- In comparison:

```
int avg_exam(struct Grades gr) {...}
```

Problem? Pass-by-value can be **expensive**, particularly for large structures.

```
int avg_exam(struct Grades & gr) {...}
```

Problem? It allows for the possibility of (mistakenly) changing the contents of the **caller's** `gr`.

Practical Use of const Reference

- One popular use of const reference: pass struct/class as the function argument

```
int avg_exam(const struct Grades & gr){
  return (gr.midterm+gr.final)/2;
}
```

- Advantages of using const reference as argument
  - We don't have the expense of a copy.
  - We have the safety guarantee that the function cannot change the caller's state.

Practical Use of const Reference

- Compared with non-const reference, another advantage is function call with consts or expressions is OK
  - In contrast, for non-const reference, function call with consts or expressions is not OK

```
foo("Hello world!")
```

```
void foo(string & str) {...}
```

versus

```
void foo(const string &str) {...}
```

const Pointers

- When you have pointers, there are two things you might change:
  1. The value of the pointer.
  2. The value of the object to which the pointer points.

- Either (or both) can be made unchangeable:

```
const T *p;   // "T" (the pointed-to object)
              // cannot be changed by pointer p
T *const p;   // "p" (the pointer) cannot be
              // changed
const T *const p; // neither can be changed.
```

pointer to const

const pointer

Pointers to const

Example

```
int a = 53;
const int *cptr = &a;
  // OK: A pointer to a const object
  // can be assigned the address of a
  // nonconst object
*cptr = 42;
  // ERROR: We cannot use a pointer to
  // const to change the underlying
  // object.
a = 28 // OK
int b = 39;
cptr = &b; // OK: the value in the pointer
           // can be changed.
```

const Pointers

Example

```
int a = 53;
int *const cptr = &a;
   // OK: initialization
*cptr = 42;
   // OK: We can use a const pointer to
   // change the underlying object.
int b = 39;
cptr = &b;
   // ERROR: We cannot change the object
   // that a const pointer points to.
```

Define Pointers to const Using typedef

- Recall typedef: give an alias to the existing types:

  `typedef existing_type alias_name;`

  - Example: `typedef int * intptr;`

    Then we can use it: `intptr ip;`

- Use `typedef` to define const pointers:
  - `typedef const T constT_t;`

    `typedef constT_t * ptr_constT_t;`
  - Now `ptr_constT_t` is an alias for the type of

    `const T *` pointer to const

Define const Pointers Using typedef

Group exercise

- Question: How do we use `typedef` to rename the type of `T *const`? const pointer

Practical Use of Pointer to const

Example

```
void strcpy(char *dest, const char *src)
  // src is a NULL-terminated string.
  // dest is big enough to hold a copy of src.
  // The function place a copy of src in dest.
  // src is not changed.
{ ... }
```

- Strictly speaking, we don't **need** to include the `const` qualifier here since the comment promises that we won't modify the source string
- So, why include it?

Practical Use of Pointer to const

Example

- Why include `const`?

- Because once you add it, you CANNOT change `src`, even if you do so by mistake.
- Such a mistake will be caught by the **compiler**.
  - Bugs that are detected at compile time are among the easiest bugs to fix – those are the kinds of bugs we want.

- **General guideline**: Use `const` for things that are passed by reference, but won't be changed.

Pointer to const versus Normal Pointer

- Pointers-to-const-T are not the same type as pointers-to-T.
- You can use a pointer-to-T anywhere you expect a pointer-to-const-T, but NOT vice versa.

```
int const_ptr(const int *ptr)
{
    …
}

int main()
{
    int a = 0;
    int *b = &a;
    const_ptr(b);
}
```

✓

```
int nonconst_ptr(int *ptr)
{
    …
}

int main()
{
    int a = 0;
    const int *b = &a;
    nonconst_ptr(b);
}
```

✗

Pointer to const versus Normal Pointer

- Why can we use a pointer-to-T anywhere you expect a pointer-to-const-T?
  - Code that expects a pointer-to-const-T will work perfectly well for a pointer-to-T; it's just guaranteed not to try to change it.

- Why **cannot** we use a pointer-to-const-T anywhere you expect a pointer-to-T?
  - Code that expects a pointer-to-T might try to change the T, but this is illegal for a pointer-to-const-T!

Reference

- Makefile
  - http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- Problem Solving with C++, 8th Edition
  - Chapter 9.1 Pointers
- C++ Primer, 4th Edition
  - Chapter 2.9 References
  - Chapter 2.4 const Qualifier
  - Chapter 4.2.5 Pointers and the const Qualifier