Ve 280

Programming and Elementary Data Structures

const Qualifier;

Procedural Abstraction

Announcement

- Project 1 released on Canvas
  - On C++ Basics
  - Due by midnight of June 4th

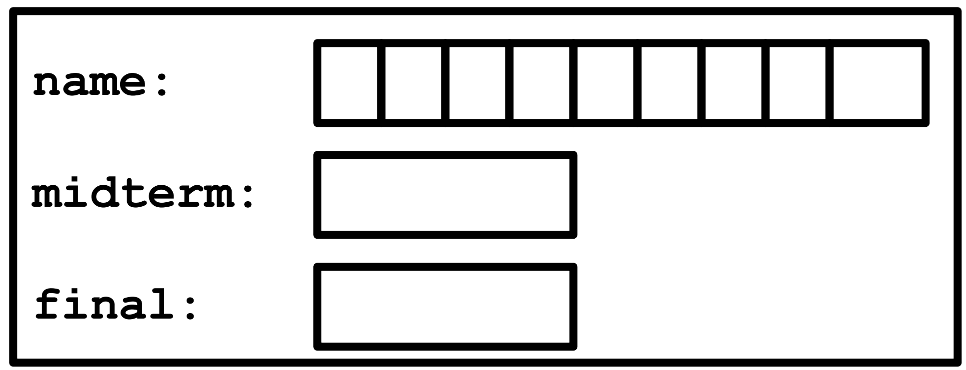
Outline

- Review of C++ Basics: Struct
- const Qualifier
- Procedural Abstraction

Structs

- Declare a `struct` type that holds grades.
- Why struct? To create a **compound type**

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

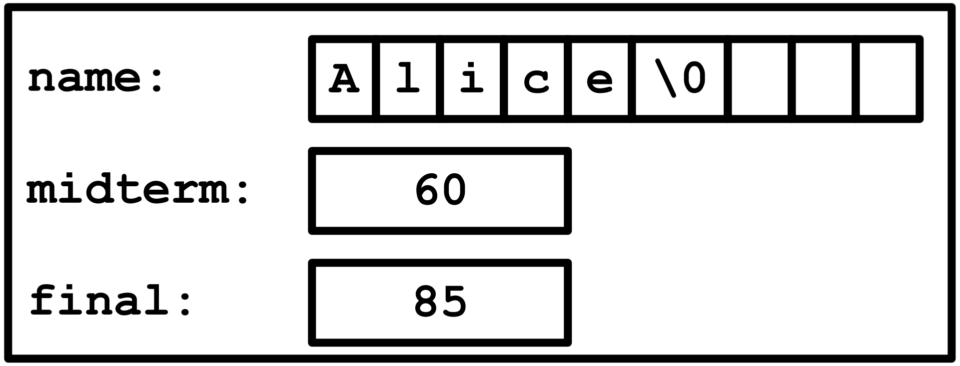


- This statement declares the **type** “struct grades”, but does not declare any **objects** of that type.
- We can define single objects of this type as follows:

```
struct Grades Alice;
```

Structs

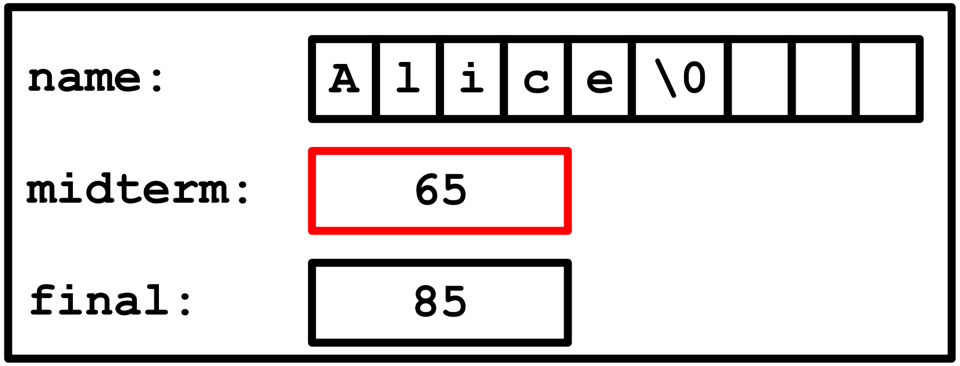
```
struct Grades {
  char name[9];
  int  midterm;
  int  final;
};
```



- 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
- If you have a pointer to struct, visit component using “`->`”

  ```
  struct Grades *gPtr = &Alice;
  gPtr->final = 90;
  ```

Outline

- Review of C++ Basics: Struct
- const Qualifier
- Procedural Abstraction

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
```

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;
}
```

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

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 value of
  // a const pointer.
```

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 pointer to const:
  - ```
    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.
- <u>Such a mistake will be caught by the **compiler**.</u>
  - Bugs that are detected at compile time are among the easiest bugs to fix – those are the kinds of bugs we want.

- <u>**General guideline**</u>: 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 you 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** you 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!

Outline

- Review of C++ Basics: Struct
- const Qualifier
- Procedural Abstraction

Abstraction

- Abstraction
  - Provides only those details that matter.
  - Eliminates unnecessary details and reduces complexity.
- Abstraction is like a black box: we know how to use a black box, but we don't know how it operates
- A person using a black box only needs to know **what** it does, NOT **how** it does it
- Example: Multiplication algorithm
  - Many ways to do: table lookup, summing, etc.
  - Each looks quite different, but they do the **same** thing.
  - In general, a user won't care how it's done, just that it multiplies.

Abstraction

- There are two types of abstraction:
  - Procedural
  - Data

Focus of this lecture

Procedural Abstraction

- **Function** is a way of providing “computational” abstractions.

```
int multi(int a, int b)
{
  // An implementation
  // of multiplication
  ...
}
```

```
int square(int a)
{
  return multi(a, a);
}
```

Using the “multi” abstraction

Procedural Abstraction

- For any function, there is a person who **implements** the function (the author) and a person who **uses** the function (the client).
- The author needs to think carefully about **what** the function is supposed to do, as well as **how** the function is going to do it.
- In contrast, the client only needs to consider the **what**, not the **how**.
- Since **how** is much more complicated, this is a Big Win for the client!
- In individual programming, you will often be the author and the client. Sometimes it is to your advantage to "forget the details" and only concentrate on abstraction.

Procedural Abstraction

- Procedural abstractions, done properly, have two important properties:
  - Local: the **implementation** of an abstraction does not depend on any other abstraction **implementation**.
    - To realize an implementation, you only need to focus **locally**.
  - Substitutable: you can replace one (correct) **implementation** of an abstraction with another (correct) one, and no callers of that abstraction will need to be modified.

Implementation of square() does not depend on **how you implement** multi()

```
int square(int a)
{
 return multi(a,a);
}
```

We can **change** the implementation of multi(). It won't affect square() as long as it does multiplication

Procedural Abstraction

- Locality and substitutability only apply to **implementations** of abstractions, not the **abstractions** themselves.
  - If you change the **abstraction** that is offered, the change is not local.
- It is CRITICALLY IMPORTANT to get the **abstractions** right before you start writing code.

```
int square(int a)
{
  return multi(a,a);
}
```

We cannot change the abstraction of "multi" to 2*a*b.

Procedural Abstraction: Summary

- Abstraction and abstraction implementation are **<u>different</u>**!
  - Abstraction: tells **what**
  - Implementation: tells **how**
  - **Same** abstraction could have **different** implementations
- If you need to change what an **abstraction** itself, it can involve many different changes in the program.
- However, if you only change the **implementation** of an abstraction, then you are guaranteed that no other part of the project needs to change.
  - **This is vital for projects that involve many programmers.**

Procedural Abstraction and Function

- **Function** is a way of providing procedure abstractions.

- The **type signature** of a function can be considered as part of the abstraction
  - Recall: type signature includes return type, number of arguments and the type of each argument.
  - If you change type signature, callers must also change.

- Besides type signature, we need some way to describe **the abstraction (not implementation)** of the function.
  - We use **specifications** to do this.

Procedural Abstraction

Specifications

- We describe procedural abstraction by specification. It answers three questions:
  - What pre-conditions must hold to use the function?
  - Does the function change any inputs (even implicit ones, e.g., a global variable)? If so, how?
  - What does the procedure actually do?
- We answer each of these three questions in a **specification comment**, and we **always** include one with **function declaration** (or function definition in case we don' have declaration)

```
…
// SPECIFICATION COMMENT
int add(int a, int b);
```

Procedural Abstraction

Specification Comments

- There are three clauses to the specification:
  - **REQUIRES**: the pre-conditions that must hold, if any.
  - **MODIFIES**: how inputs are modified, if any.
  - **EFFECTS**: what the procedure computes given legal inputs.

- Note that the first two clauses have an "if any", which means they may be empty, in which case you may omit them.

Procedural Abstraction

Specification Comment Example

```
bool isEven(int n);
    // EFFECTS: returns true if n is even,
    // false otherwise
```

- This function returns true if and only if its argument is an even number.
- Since the function isEven is well-defined over all inputs (every possible integer is either even or odd) there need be no REQUIRES clause.
- Since isEven modifies no (implicit or explicit) arguments, there need be no MODIFIES clause.

Procedural Abstraction

Specification Comment Example

```
int factorial(int n);
    // REQUIRES: n >= 0
    // EFFECTS: returns n!
```

- The mathematical abstraction of factorial is only defined for non-negative integers. So, there is a **REQUIRE** clause.
- The **EFFECTS** clause is only valid for inputs satisfying the **REQUIRES** clause.
- Importantly, this means that the implementation of factorial DOES NOT HAVE TO CHECK if n < 0! The function specification tells the caller that s/he **must** pass a non-negative integer.

Procedural Abstraction

More Function Details

- Functions without REQUIRES clauses are considered **complete**; they are valid for all input.
- Functions with REQUIRES clauses are considered **partial**
  - Some arguments that are "legal" with respect to the type (e.g., int) are not legal with respect to the function.
- Whenever possible, it is much better to write complete functions than partial ones.

- When we discuss **exceptions**, we will see a way to convert partial functions to complete ones.

Procedural Abstraction

More Function Details

- What about the MODIFIES clause?

- A MODIFIES clause identifies any function argument or global state that **might** change if this function is called.
  - For example, it can happen with call-by-reference as opposed to call-by-value inputs.

Procedural Abstraction

Specification Comment Example

```
void swap(int &x, int &y);
  // MODIFIES: x, y
  // EFFECTS: exchanges the values of
  // x and y
```

- NOTE: If the function **could** change a reference argument, the argument must go in the MODIFIES clause. Leave it out only if the function can **never** change it.

Reference

- const Qualifier
  - C++ Primer, 4th Edition, Chapter 2.4
- const Pointers
  - C++ Primer, 4th Edition, Chapter 4.2.5
- const References
  - C++ Primer, 4th Edition, Chapter 2.5
- Procedural Abstraction
  - Problem Solving with C++, 8th Edition, Chapter 4.4 and 5.3